Chapter 1 Introduction

1.1 History of the eP32

The eP32 microprocessor is a Minimal Instruction Set Computer (MISC), vis-à-vis Complicated Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC). MISC was originally developed by Mr. Chuck Moore, and implemented in his MuP21 chip. It happened that Chuck also invented the FORTH programming language. For many years, Chuck sought to put FORTH into silicon, because he thought FORTH was not only a programming language, but also an excellent computer architecture.

In the early 1990s, a group of engineers from the MOSIS multiple design chip service program came to Silicon Valley and started Orbit Semiconductor Corp, offering foundry services to the general public. Their service was based on a 1.2 micron CMOS processes on 5 inch wafer, with two metal layers. The smallest design they accepted was on a 2.4mmx2.4mm silicon die. Chuck figured that he could design a 20 bit CPU in that small area. It was named MuP21, because it was a multiprocessor chip, with a 20 bit CPU core, a DRAM memory coprocessor, and a video coprocessor, and all registers and stacks in the CPU core were 21 bits wide, with an extra bit to preserve the carry bit.

Because of very limited silicon area, the MuP21 had a very small set of instructions, but they were sufficient to support a complete FORTH operating system and very demanding applications with real time NTSC video output. The chip was produced and verified, but productions in plastic packages were not successful because of poor yield.

When FPGA chips became available, I tried to implement FORTH chips based on MuP21 instruction set. The first experiments were on an XS40 Kit from Xess Corp. It had a Xilinx VC4005XL FPGA on board with a 32 kB SRAM chip and an 8051 microcontroller. The purpose of this kit was to demonstrate how easy it was to use an FPGA to replace all glue logic between RAM and 8051, and to build a complete working microprocessor system. I managed to squeeze a 16-bit microprocessor, P16, into the VC4000XL chip and eliminated the 8051.

Over the years, Xilinx added more logic gates and RAM blocks to their FPGAs, and I was able to put a 32-bit microprocessor, P32, into a VCX1000E chip (which had 16 kB of RAM) to host a FORTH system. This design was also ported to FPGA chips from Altera and Actel. P32 gradually evolved into eP32 with an eForth operating system. eForth is a very simple FORTH operating system designed specifically for embedded systems. However, FPGA chips were expensive, development boards were expensive, and development software tools were especially expensive. I talked about eP32 implementations, but very few people in the audience had these development tools to explore FPGA designs.

It was therefore very exciting to learn about the LatticeXP2 Brevia Development Kit, which was on sale for $49. Development software was free to download. The Kit has a LatticeXP2-5E-6TN144C FPGA chip, which has enough logic cells to

implement eP32, and enough RAM memory to host the eForth system. Its RAM memory is mirrored in flash memory on chip, and you do not need external memory chips for programs and data. It is truly a single chip solution for microprocessor system design.

Now, everybody can do his own designs on FPGA chips. It is time to update my documentation on eP32 and companion eForth to teach people the best way to design their own CPUs and to explore their applications.

All FPGA manufacturers offer reference designs of microprocessors in their development software tools, to demonstrate that FPGAs can be used to do microprocessor system designs, or in a fancier term System-On-a-Chip, SOC. However, these microprocessors are complicated, and their performance is poor. A microprocessor does not work without software. Software reference designs from these FPGA manufacturers are even poorer, as we see them struggle with assemblers, language compilers, and operating systems.

FORTH offers the best solution for FPGA users. The CPU is simple, the programming language is simple, the operating system is simple, and the application programming is simple. It is possible for an average engineer or scientist to understand and to make use of this complete CPU-Language-Operating System-Application spectrum in a few weeks. What's required is an open mind, and a willingness to explore different ways to do things. The very high cost barrier to experiment with an FPGA is removed by the LatticeXP2 Brevia Kit. The only barrier left is you yourself.

This book contains two major sections, one on hardware design of the eP32 CPU core and a few peripheral devices to form a complete microprocessor, and one on the software design of eForth to run on the eP32. Hardware design is centered on a set of VHDL files, describing modules in the eP32 microprocessor system. Software design is centered on a set of FORTH files, which is a metacompiler constructing a memory image to initialize a RAM memory module in the eP32. Generally, I will show source code on left hand pages, and commentary on the opposing right hand pages. My perspective is that source code is supreme. Nothing is more important than source code. If you understand the complete source code, you understand everything.

Combining the hardware design of the eP32 and software design of eForth, the result is a FORTH microprocessor running on a LatticeXP2 Brevia Development Kit. You can run this FORTH microprocessor from a HyperTerminal console on your PC, and write application programs. Mastering this book, you have an understanding of one microprocessor, in and out. This understanding will allow you to develop your own microprocessor to solve your own application problems.

The eP32 has a 32-bit CPU core with two stacks. It was intended to execute FORTH instructions efficiently. The processor design is simple to allow implementation on custom silicon chips as well as on FPGAs. The eP32 employs only 27 instructions, and instruction can be encoded in 5 bit fields. This design is scalable in word sizes ranging from 16 bits up to 64 bits. A program word can contain many instructions in 5 bit fields. With this scalable architecture, a CPU designer is freed from the heavy

yoke of program word size, which is a primary constraint on a CPU design.

1.2 What is FORTH?

FORTH was invented by Chuck Moore in the 1960s as a programming language. Chuck was not impressed by programming languages, operating systems, and microprocessor hardware of his time. He sought the simplest and most efficient way to control his computers. He used FORTH to program every computer in his sight. And then, he found that he could design better computers, because FORTH is much more than just a programming language; it is an excellent computer architecture.

So what is FORTH?

Many books and many papers had been written about FORTH. However, FORTH is still elusive because it has many features and characteristics which are difficult to describe. Now that it has erased the boundary between hardware and software, it is even more difficult to accurately put it into words.

Let me try this way. Here it goes.

FORTH is a list processor.

FORTH has a set of commands, and an interpreter to process lists of commands.

FORTH commands are records stored in a memory area called a dictionary.

A record of a FORTH command has three fields: a link field linking commands to form a dictionary, a name field containing the name of this command in an ASCII string, and a code field containing executable code and data to perform a specific function for this command. It may have an optional parameter field, which contains data needed by this command. The link field and name field allow the interpreter to look up a command in the dictionary, and the code field provides executable code to perform the function assigned to this command.

A FORTH command has two representations: an external representation in the form of an ASCII name; and an internal representation in the form of a token, which invokes executable code stored in code field. In many FORTH systems, the token is an address. However, a token can take other forms depending on implementation.

There are two types of FORTH commands: primitive FORTH commands having machine code in their code fields, and compound FORTH commands having token lists in their code fields.

A FORTH interpreter processes two types of lists: text lists and token lists. A text list contains a sequence of FORTH command names, separated by white spaces and terminated by a carriage return. A token list contains a sequence of tokens, which are internal representations of FORTH commands.

FORTH has two interpreters: a text interpreter (or outer interpreter) and a token interpreter (or inner interpreter).

The text interpreter processes lists of FORTH commands represented in text, which consists of names of FORTH commands separated by white spaces and terminated by a carriage return. The number of commands in a text list is not limited. A list may be in one line of text, or in a huge text file.

The token interpreter processes lists of tokens contained in compound commands. It is also called the address interpreter, because in many FORTH systems, tokens are addresses pointing to code fields.

The text interpreter operates in two modes: interpreting mode and compiling mode. In interpreting mode, a list of command names is interpreted; i.e., commands are parsed and executed. In compiling mode, a list of command names is compiled; i.e., commands are parsed and corresponding tokens are compiled into a token list. This token list can be given a name to form a new compound command, by creating a new command record in the dictionary.

A FORTH compiler is a FORTH text interpreter operating in compiling mode. It compiles new compound commands, converting a text list of FORTH commands into an equivalent token list. It builds nested token lists one on top of the other, until a final solution is reached in the last token list.

This is the most powerful feature of FORTH, in that you can compile new compound commands, which replace lists of existing commands, both primitive and compound. The syntax of a new compound command is:

```
: <name> <list of existing commands> ;
```

A FORTH compiler converts a text list of existing commands to a new token list. Nested token lists are added until the final compound command becomes the solution to your problem. Lists are built and tested from the bottom up. The solution space can be explored wider and farther, and an optimized solution can be found more quickly.

Following are some minor deviations in the syntax of FORTH as a programming language.

The text interpreter accepts numbers in lists. Numbers are ASCII strings with valid numeric digits and an optional leading '-' sign. The text interpreter pushes an integer number onto the data stack. The FORTH compiler compiles an integer literal into the token list. Later, when the token list is interpreted, the integer literal token pushes the integer onto the data stack.

The text interpreter accepts strings in lists. A string must follow a string command, which consumes the string. A string is a sequence of ASCII characters terminated by a terminating character specified by the preceding string command. A string command may compile a string literal into the token list. In the token list, a string literal consists of a string token followed by the string in compiled form. The string token uses the compiled string, and passes control to the next token after the compiled string.

Lists are normally processed in consecutive sequence. However, branches and loops

are allowed, using control structure commands. Control structure commands compile control structures into token lists. Later, when a token list is interpreted, branching and looping occur within those control structures.

String commands and control structure commands change sequential flow in lists. They are elements in the FORTH language that require additional grammatical rules in their usage. Otherwise, all lists are simple, linear, sequential lists.

The preceding exposition describes what FORTH is in terms of a programming language and operating system. A complete specification of a FORTH system must include a document on all commands; i.e., names of commands, their effects on data and return stacks, and their functional descriptions.

The fundamental reason that FORTH lists can be simple, linear sequences of commands is that FORTH uses two stacks: a return stack to stored nested return addresses, and a data stack to pass parameters among nested commands. Parameters are passed implicitly on the data stack, and do not have to be explicitly invoked. Therefore, FORTH commands can be interpreted in a linear sequence, and tokens can be stored in simple, linear lists. Language syntax is greatly simplified, internal representation of tokens is greatly simplified, and execution speed is greatly increased.

A FORTH CPU thus needs two stacks, efficient means to traverse nested token lists, and an instruction set to support primitive commands. This is what eP32 is designed to provide. It has two stacks. It has a small instruction set, which is sufficient to code all primitive commands in eForth. It has very efficient single cycle subroutine call and return instructions. When we use the Subroutine Threading Model (where a compound command consists of a list of subroutine call instructions) and represent tokens by subroutine call instructions, the eP32 CPU itself becomes the FORTH inner interpreter. Nested token lists, as nested subroutine lists, are traversed naturally with very little overhead in execution speed.

The eP32 is the best list processor.

Chapter 2. Design of the eP32

2.1 Overview

The eP32 is a 32-bit CPU. Instructions are encoded in 6-bit fields, and up to 5 instructions are packed into a single 32-bit program word. 27 instructions are defined to facilitate accessing words in memory, for multiplication and division of integers, for real time interrupts and to support various IO devices. A return stack is included in the CPU for nested subroutine calls and returns. A parameter stack is also included to pass parameters among nested subroutines. The simple instruction set and dual stack design make it possible to execute all instructions in a single clock cycle from a single phase master clock. This design optimizes code density, processing speed, silicon area and power consumption, and is most suitable to serve as CPU cores in System-On-a-Chip integrated circuits.

As this design was developed and tested on a large FPGA device, the LatticeXP2 from Lattice Semiconductor Corp, a complete microprocessor system, including CPU, memory and a number of I/O devices, is built on a single FPGA chip.

In this design, the CPU latches all data into appropriate registers and stacks on the rising edge of a single phase master clock. Such a synchronous design ensures that all instructions are executed quickly and reliably in a single clock cycle. When the master clock is held steady, the microprocessor retains all data in registers, stacks and memory, consuming very little power. It is thus possible to further reduce its power consumption by reducing the clock rate, or stopping the clock completely.

The eP32 has this set of registers:

Name	Register	Function
I	Instruction latch	Holding up to 5 instructions to be executed
P	Program counter	Pointing to next program word in memory
R	Top of return stack	Holding return address or loop counts
S	Second item of data stack	Supplying optional second argument to ALU
T	Top of data stack	Accumulator for ALU
X	Address register	Supplying address for memory read and write

The eP32 has two stacks to support fast subroutine calling and returning, and to optimize execution speed:

Name	Stack	Function
s_stack	Data stack	Passing arguments among nested subroutines
r_stack	Return stack	Saving return addresses of nested subroutines

The I and P registers are 32 bits wide to address 4G words of memory. T, R, S, X and stacks are all 33 bits wide. The most significant bit in T, T(32) is a carry produced by a 32-bit adder. This carry bit is preserved when T is transferred to other registers and to stacks. Preservation of carry bit greatly simplifies extended precision arithmetic operations in the ALU, and allows subroutines and interrupts to be serviced without having to save a carry bit and restore it on return.

Registers and stacks and their relationship are best shown in Figure 1:

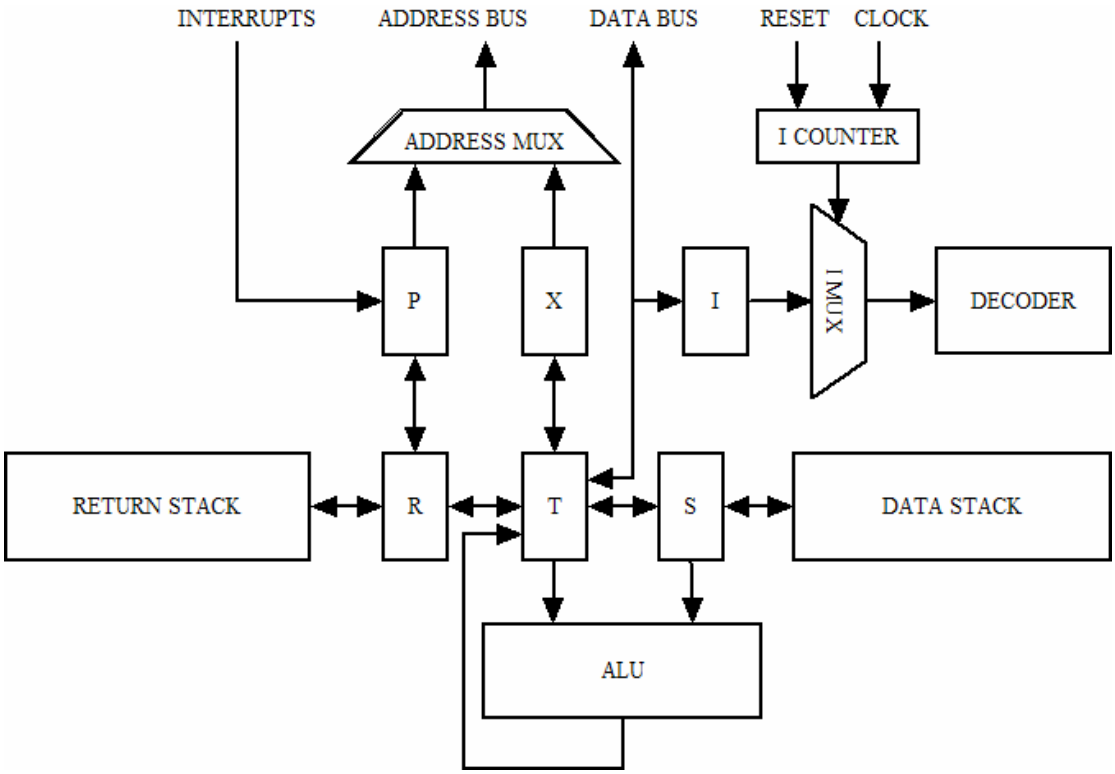

Figure 1. eP32 Architecture

The T register is the center of the eP32. It supplies one argument to the ALU, which takes an optional second argument from the S register and routes results back to the T register. Contents in T can be moved to the X register, pushed on data stack S, or pushed on return stack R.

The T register connects data stack and return stack as a giant shift register. Data can be shifted towards the return stack by a PUSH instruction, and shifted towards the data stack by a POP instruction.

Register X holds a memory address, which is used to read data from memory into the T register, or write data from the T register to memory. The address in X can be auto incremented, so that the eP32 can conveniently access data arrays in memory.

P is a program counter and holds the address of the next program word to be fetched from memory. After a program word is fetched, P is auto incremented and ready to read the next word. When a CALL instruction is executed, the address in P is pushed onto the return stack. When a RET return instruction is executed, the previously saved address on the return stack is popped back into P. The execution sequence interrupted by CALL can then be resumed.

The depth of both stacks is 32 levels, which allows very deep nesting of subroutine calls. Stacks are implemented as circular buffers. Overflow and underflow

overwrite data previously pushed onto the stack 32 levels before. No effort is made in detecting and handling overflow and underflow conditions, because stack overflow/underflow is not really a very serious error condition, although it is dreaded by programmers using conventional languages. Commands may consume stack items, and may push data onto a stack. It is impossible for an operating system to determine whether the stack effects of a command are due to errors or due to the programmer's intention. Therefore, it is best left to the programmer to make sure that stacks behave correctly.

The 6-bit code field supports up to 64 instructions. Five 6-bit instructions are packed into one 32-bit word, and are executed consecutively after a program word is fetched from memory. It can be viewed as a 5 instruction cache, which provides an optimal balance between a slow RAM memory and a fast CPU. For example, if 32-bit words can be fetched from RAM at a rate of 20 MHz, the 5 instructions can be executed at a rate of 100 MHz.

The design and functions of the eP32 are best presented in functional blocks. The eP32 can be divided into the following 4 functional blocks, in four quadrants of the above diagram:

Program Execution Unit in Quadrant 1
Memory Address Multiplexer in Quadrant 2
Return Address Processing Unit in Quadrant3
Data Processing Unit in Quadrant 4

These blocks will be discussed in the following pages.

2.2 Program Execution Unit

A synchronous Program Execution Unit is a finite state machine, controlling execution of instructions in the eP32. It has a COUNTER register driven by external "reset" and "clock" signals. When "reset" is asserted, COUNTER is cleared to 0, which is output to "slot". When "reset" is released, external clock signal "clock" drives COUNTER, which is incremented on the rising edge of "clock". "slot" is incremented from 0 to 5, and back to 0. When "slot"=0, eP32 reads the next program word from the Data Bus, and latches it into the I register on the rising edge of "clock".

As "slot" is incremented between 1 and 5, it selects from the I register one 6-bit instruction "code" through instruction multiplexer IMUX. "code" drives DECODER, which produces all control signals to run the eP32. These control signals select appropriate data through multiplexers, and present them to registers and stacks. On the rising edge of "clock", selected data are latched into appropriate registers and stacks, and thus starts another instruction cycle.

When executing transfer instructions like CALL, BRA, BZ, BC, NEXT, RET and NOP, the "slot0" signal is set. It clears COUNTER and forces next cycle back to slot0, fetching a new program word from the Data Bus.

The rising edge of the "clock" signal thus paces the eP32 to execute instructions read from external memory through the Data Bus. The eP32 is a synchronous CPU.

Registers and stacks are changed only on the rising edge of “clock”. Otherwise, all registers and stacks are static, and hold their contents indefinitely.

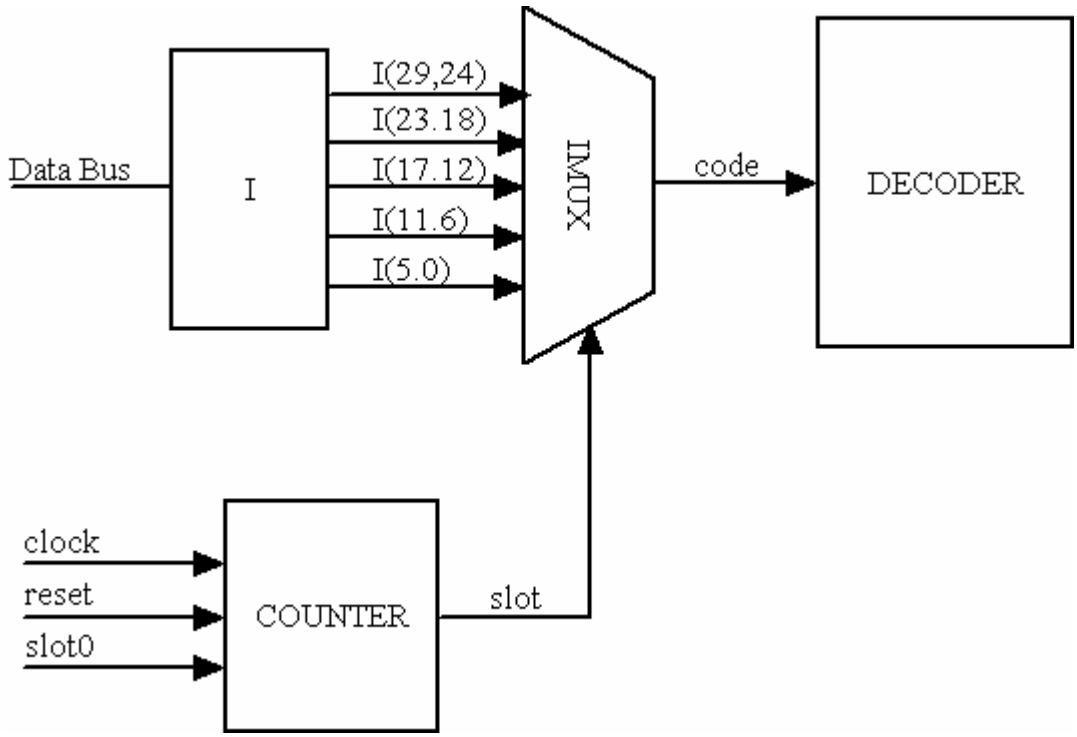

Figure 2. Program Execution Unit

2.3 Memory Address Multiplexer

The Memory Address Multiplexer supplies a 32-bit address on the Address Bus to external devices. When executing the next program word, the AMUX multiplexer routes the address stored in the P register to the Address Bus. When accessing data in memory, the XMUX multiplexer routes the address stored in the X register to the Address Bus. This symmetrical arrangement of P and X registers and address multiplexers AMUX/XMUX allows all memory operations to be completed in a single machine cycle. This is the simplest memory management system of a von Neumann machine. It is entirely unnecessary to use very complicated memory modes to access memory, as in CISC computer designs.

Depending on the current instruction being executed, PMUX selects one of 4 inputs to the P register: the next program address (P+1), a target address in the address field of the current program word in the I register, the return address in the R register, and an interrupt vector. The selected new address is latched into the P register on the rising edge of master clock.

Depending on the current instruction being executed, XMUX selects one of 5 inputs to the X register: the T register, the next data word address (X+1), the left-shifted (T+S):X register pair in a divide step instruction, the right-shifted T:X register pair in multiply step instruction, and the (T+S):S register pair in a multiply step instruction. Selected new data is latched into the X register on the rising edge of the master clock.

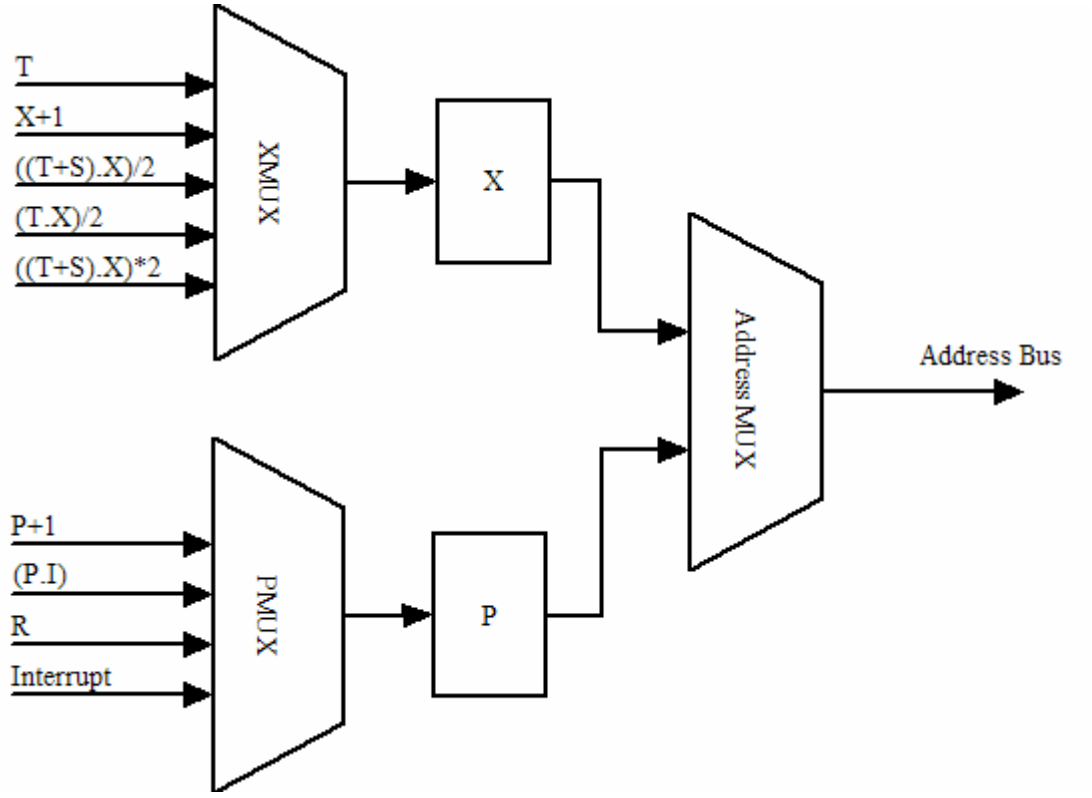

Figure 3. Address Unit

2.4 Data Processing Unit

The Data Processing Unit contains a data stack and an ALU. The top item of data stack is implemented as the T register, which is like an accumulator in conventional CPU designs. The top element of data stack is designated as the S register. The ALU takes T and S registers as its input and generates a set of logic and arithmetic signals. TMUX selects one of these results and routes it to the T register. A specific machine instruction will select the result it needs and latch it into the T register on the rising edge of the master clock. This strategy—Compute Everything and Select the One You Need—allows all ALU operations to be complete in a single machine cycle.

All ALU instructions select the results they want through TMUX. You can recognize these instructions by the signals in front of TMUX.

The PUSH instruction selects the S register to load the T register. The POP instruction selects the R register to load T. The XT instruction selects X to load T. Memory read instructions select the Data Bus to load T.

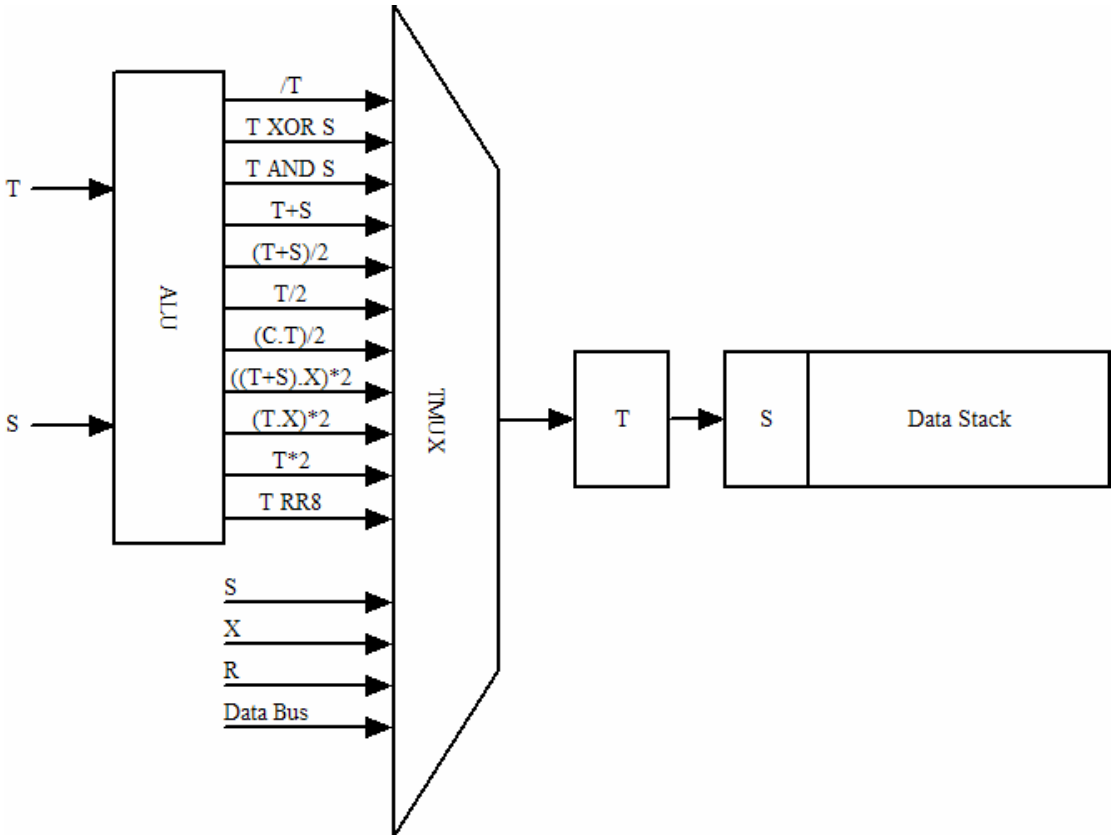

Figure 4. Data Processing Unit

2.5 Return Address Processing Unit

The Return Address Processing Unit allows subroutine CALL and RET instructions to be executed in a single machine cycle. It contains a return stack, whose top item is implemented as the R register. A CALL instruction pushes the address of the next program word in the P register onto the return stack through RMUX. A RET instruction pops the return stack and latches the return address in R back into the P register.

Subroutine call and return instructions generally are the most complicated machine instructions in a CISC computer design. They all take many clock cycles to complete, because many tasks are required in nesting and un-nesting a subroutine call. Here in the eP32, subroutine call and return are both reduced to a single clock cycle. As all compound programming languages rely heavily on subroutine calls and returns, reducing overhead in subroutine calls and returns will significantly improve performance of programs produced by these language compilers.

The eP32 is also optimized to process loops. During looping, the R register is used to hold a loop count. The NEXT instruction looks at this count. If R is not zero, NEXT decrements it and branches to the beginning of the loop. If R is zero, NEXT terminates the loop. To decrement R, R-1 is selected by RMUX to latch back into R on the rising edge of the master clock.

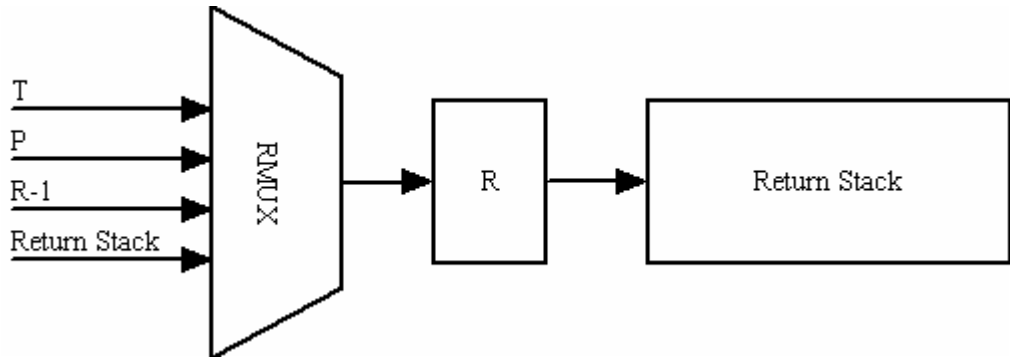

Figure 5. Return Stack Unit

2.6 Timing of Instruction Execution

This simple yet efficient design of the eP32 allows all instructions to be executed in a single clock cycle. Each machine clock cycle is called a "slot". However, program words must be read into the CPU before instructions in them can be executed. In the current implementation, I allocate an extra cycle to read in a program word. This extra cycle is called "slot0". After a program word is read in "slot0", as many slots are used to execute as many machine instructions in the program word as necessary. For short instructions, 1 to 5 more slots are used to execute 1 to 5 instructions. For long instructions, only "slot1" is used to execute a single long instruction in a program word.

The following diagram shows timing in executing short instructions and long instructions.

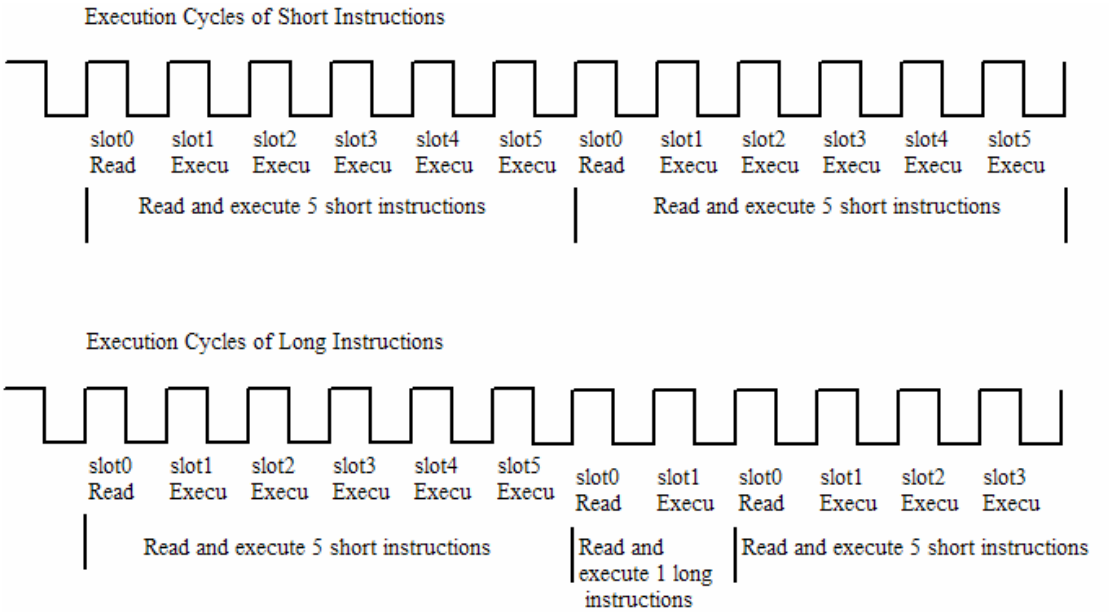

Figure 6. Instruction Exection Timing

NOP and RET instructions can be in any of the 5 slots in a program word. When these two instructions are executed, "slot0" will be the next slot, and the next program word will be fetched from memory and then executed. Extra NOP instructions filled

in a program word by a compiler do not waste extra clock cycles.

Under most circumstances, fetching the next program word can be overlapped with other machine instructions, and “slot0” can be buried to save execution time. However, an explicit “slot0” to fetch the next program word allows servicing real time interrupts with very little extra hardware overhead. In “slot0”, interrupt pins are examined. If not all interrupt pins are 0 and interrupts are enabled, the non-zero 5-bit pattern presented by the interrupt pins are taken as the address of a subroutine call, and execution is transferred to one of the locations between 1 and 31. In memory, Location 0 contains the reset vector, and locations 1-31 contain 31 interrupt vectors.

Interrupt is a big issue in microprocessor designs. If you are familiar with early microprocessors, you might remember that the 8089 interrupt controller in the 8080 microprocessor family was as complicated as the 8080 itself. Here I provide a very simple solution. It is not a “be all, do all” solution for interrupts, but it gives you something to start with.

Chapter 3 ePP32 Instructions

3.1 Instruction Classes

The eP32 executes a small but comprehensive set of machine instructions. There are two types of machine instructions. A long instruction has the following format, with a 6-bit instruction field and a 24-bit address field:

00	cccccc	aaaaaa	aaaaaa	aaaaaa	aaaaaa

When executing a long instruction, the lower 24-bits in the P register are replaced by the contents of the address field so that the next program word will be fetched from a new address. Long instructions have 24-bit address fields, which allow branching inside a 16M-word memory page. If you have to jump to an address outside of the current memory page and in the full 32-bit addressing space of 4G words, you must first load a 32-bit address into the T register, push it on return stack, and then execute a RET instruction. This method allows you to jump to any memory location.

The short instructions are 6-bit in width, and 5 such instructions can be packed into one program word as shown in the following format:

00	cccccc	cccccc	cccccc	cccccc	cccccc

The top two bits in a 32-bit program word are not used. Experienced CPU designers will find these bits useful in extending the instruction set of the eP32 CPU.

As an instruction code of the eP32 has 6 bits, there can be 64 instructions. We have defined only 27 instructions in the eP32, leaving plenty of room for sophisticated designers to add custom instructions for specific applications.

The complete instruction set is shown in Appendix A for your reference.

eP32 instructions can be divided into five classes:

Instruction Class	Instructions
Transfer Instructions	BC, BRA, BZ, CALL, NEXT, RET
Memory Access Instructions	LDI, LDX, LDXP, STX, STXP
ALU Instructions	ADD, AND, COM, DIV, MUL, RR8, SHL, SHR, XOR
Register/Stack Instructions	DROP, DUP, NOP, OVER, POP, PUSH, SWAP, TX, XT
Miscellaneous Instructions	EI

Transfer instructions BC, BRA, BZ, CALL and NEXT are long instructions with a 24-bit address field. These instructions allow a program to branch to a new location inside the current page of memory. A page is 16M words in size. The current page is where the current program word resides.

Names, binary code and function of these instructions are listed below, sorted by instruction code.

Instruction	Code	Function
BRA	000000	Branch to address contained in address field.
RET	000001	Return from a subroutine to calling program. Pop return address from return stack and deposit it in P.
BZ	000010	If T=0, branch to address in address field; else continue.
BC	000011	If Carry is 1, branch to address in address field; else continue.
CALL	000100	Push address in P on R stack, and branch to address in address field.
NEXT	000101	If R is not 0, branch to address in address field, and decrement R by 1; else pop R stack and continue.
EI	000110	Enable interrupts.
LDXP	001001	Push T on S stack; read data word pointed to by X into T. Increment X by 1.
LDI	001010	Push T on S stack; read data word pointed to by P into T. Increment P by 1.
LDX	001011	Push T on S stack; read data word pointed to by X into T.
STXP	001101	Store T into word pointed to by X. Increment X by 1. Pop S stack to T.
RR8	001110	Rotate T right by 8 bits.
STX	001111	Store T into word pointed to by X. Pop S stack to T.
COM	010000	Complement T (1’s complement).
SHL	010001	Shift T left by 1 bit.
SHR	010010	Shift T right by 1 bit.
MUL	010011	Multiplication step. If X(0)=1, add S to T. Shift T:X pair right by 1 bit.
XOR	010100	Pop S stack and XOR it to T.
AND	010101	Pop S stack and AND it to T.
DIV	010100	Division step. If T+S produces a carry, shift (T+S):X pair left by 1 bit and set X(0); else shift T:X left by 1 bit.
ADD	010111	Pop S stack and add S to T.
POP	011000	Push T onto S stack. Pop R stack to T.
XT	011001	Push T onto S stack. Copy X to T.
DUP	011010	Push T onto S stack. T remains unchanged.
OVER	011011	Push T onto S stack. Copy original contents of S to T.
PUSH	011100	Push T onto R stack. Pop S stack to T.
TX	011101	Copy T to X. Pop S stack to T.
NOP	011110	No operation.
DROP	011111	Pop S stack to T.

All other instructions are short 6-bit instructions. Up to 5 short instructions can be packed in to a single 32-bit program word. However, when the RET instruction is executed, execution is transferred to the address saved on the return stack, and subsequent short instructions in the same program word are ignored. NOP behaves similarly so that extra NOP instructions filled in by the compiler are ignored.

In many instances, a program word cannot be filled with useful short instructions, because the next instruction is a long instruction, and the rest of the current program word must be filled with NOP instructions. Instead of wasting time to execute these NOP instructions, the instruction sequencer in eP32 will abandon the current program word, immediately fetch the next program word and execute it when it encounters the first NOP instruction. However, the user does not have to worry about this, because the compiler automatically packs as many short instructions into a program word as possible. Only when the compiler must start a long transfer instruction does it fill the current program word with NOPs.

3.2 Transfer Instructions

Instruction	Code	Function
BC	000011	If Carry is 1, branch to address in address field; else continue.
BRA	000000	Branch to address in address field.
BZ	000010	If T=0, branch to address in address field; else continue.
CALL	000100	Push the address in P on R stack, and branch to address in address field; else continue.
NEXT	000101	If R is not 0, branch to address in address field, and decrement R by 1; else pop R stack and continue.
RET	000001	Return from a subroutine to calling program. Pop return address from return stack and deposit it in P.

BRA is an unconditional branch instruction. It branches to a location in the current memory page of 16M words. BZ is the branch on zero instruction. It branches to a new location when the lower 32 bits in T are all 0. Otherwise it is a NOP. It is used extensively in FORTH to construct IF-ELSE-THEN branch structures, and BEGIN-UNTIL and BEGIN-WHILE-REPEAT loop structures.

BC is the branch on carry instruction. It branches to a new location if the carry bit produced by adder in the ALU is set. Otherwise it is a NOP. This instruction is not used in compound commands, but is used to implement many primitive commands where extended precision integer arithmetic operations require a carry bit.

CALL and RET are used to do subroutine nesting and unnesting. The eForth software system uses a Subroutine Threading Model. All compound commands are defined as subroutines.

The NEXT instruction reduces a looping operation to a single cycle instruction. In eForth, one enters a FOR-NEXT loop structure by pushing a loop count into the R register. By adding auto-decrement and zero-detect functions to the R register, it is possible to implement NEXT in hardware as a single cycle machine instruction, and

thus optimize counted looping operations in eForth.

3.3 Memory Access Instructions

Instruction	Code	Function
LDI	001010	Push T on S stack, read data word pointed by P into T. Increment P by 4.
LDX	001011	Push T on S stack, read data word pointed by X into T.
LDXP	001001	Push T on S stack, read data word pointed by X into T. Increment X by 1.
STX	001111	Store T into memory pointed by X. Pop S stack to T.
STXP	001101	Store T into memory pointed by X. Increment X by 1. Pop S stack to T.

The P-series microprocessor addresses memory in words of whatever the width is of program and data words. The eP32 instruction set assumes 32-bit addresses and 32-bit program and data words. It does not address bytes in memory.

The LDI instruction reads the next word in program memory and pushes it on the data stack. The word address is in the P register. The P register is auto-incremented to skip the data word. LDI allows literal integers to be stored in programs and read into the CPU at run time. Literal integers are very important constituents of programs, and LDI instructions optimize their storage and usage.

The LDX instruction loads a 32-bit word from memory to the T register. STX stores the 32-bit word that is in the T register to a word location in memory. The memory address is in the X register.

LDXP and STXP are like LDX and STX, respectively, except that after memory access, the X register is auto-incremented. Auto-incrementing the X register allows consecutive memory locations to be read or written with minimal overhead.

3.4 ALU Instructions

Instruction	Code	Function
ADD	010111	Pop S stack and add it to T.
AND	010101	Pop S stack and AND it to T.
COM	010000	Complement T (1’s complement).
DIV	010100	Division step. If T+S produces a carry, shift the (T+S):X pair left by 1 bit and set X(0); else shift T:X left by 1 bit.
MUL	010011	Multiplication step. If X(0)=1, add S to T. Shift the T:X pair right by 1 bit.
RR8	001110	Rotate T right by 8 bits.
SHL	010001	Shift T left by 1 bit.
SHR	010010	Shift T right by 1 bit.
XOR	010100	Pop S stack and XOR it to T.

In the original MuP21 design, only COM, SHL, SHR, AND, XOR, and ADD

instructions were defined. Other logic and arithmetic operations were implemented in terms of these basic instructions. In the eP32, MUL, DIV and RR8 are added.

COM, SHL, SHR, and RR8 are unary operations on the T register alone.

COM does one's complement on T register. SHL shifts the T register 1 bit to the left. SHR shifts T register 1 bit to the right.

RR8 rotates the contents of the T register to the right by 8 bits. This instruction is very useful in a word-addressing CPU like the eP32. It allows individual bytes in memory to be accessed with minimal effort.

ADD, AND and XOR are binary operations on the T and S registers. They pop the data stack and discard the data in the S register.

ADD adds S to T. AND ands S to T. XOR exclusive ors S to T.

OR is not implemented as a machine instruction. It is implemented in software using De Morgan's theorem. In many cases, XOR can be used to perform OR functions.

MUL and DIV are ternary operators, involving the T, S and X registers. MUL is a muliply step instruction and DIV is a divide step instruction.

Multiplication and division are important arithmetic operations frequently used in computation-intensive applications. It is possible to implement a full multiplier-adder for DSP applications. However, a fast multiplier-adder requires a large number of gates and significantly increases power consumption. In the eP32, a multiplication step instruction, MUL, and a division step instruction, DIV, are implemented. They make use of the 32-bit adder and shifter already existing in the ALU. Very little hardware is added, and very little additional power is needed.

In the MUL instruction, the T and X registers are considered a 65-bit right-shift register. Initially, a partial sum is loaded in the T register, a multiplier in the X register, and a multiplicand in the S register. If the least significant bit in X is 1, S is added to T, and the resulting T-X pair is shifted right by 1 bit. If the least significant bit in X is 0, T is not changed, and the T-X pair is shifted right by 1 bit. This MUL instruction is repeated 32 times, after which the T-X register pair will contain a double-word product of X*S +T. The MUL instruction is shown in the following diagram:

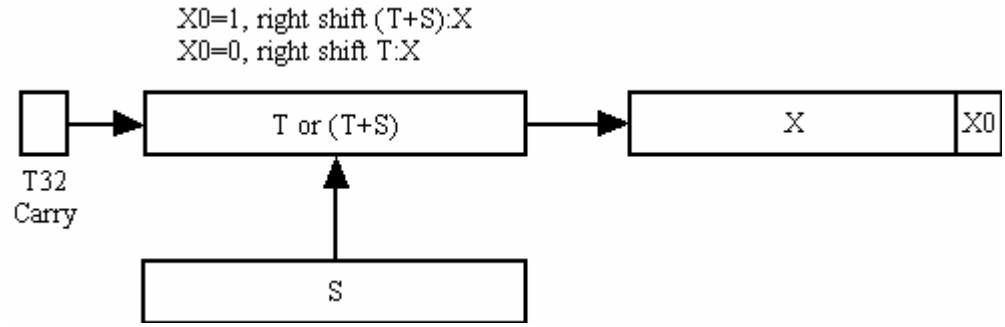

Figure 7. Mulitlication Step

In the DIV instruction, the T and X registers are considered a 65-bit left-shift register. A double integer dividend is in the T-X register pair, and a negated divisor is in the S register. In the ALU, the sum of S and T is always computed by an adder. If the carry bit in the adder is 1, S is added to T, and the resulting T-X pair is shifted left by 1 bit. If the carry bit in the adder is 0, T is not changed, and the T-X register pair is shifted left by 1 bit. In either case, the carry bit is shifted into the least significant bit in the X register. After repeating the DIV instruction 32 times, the X register contains quotient, and the T register contains 2x of the remainder of the division. The DIV instruction is shown in the following diagram:

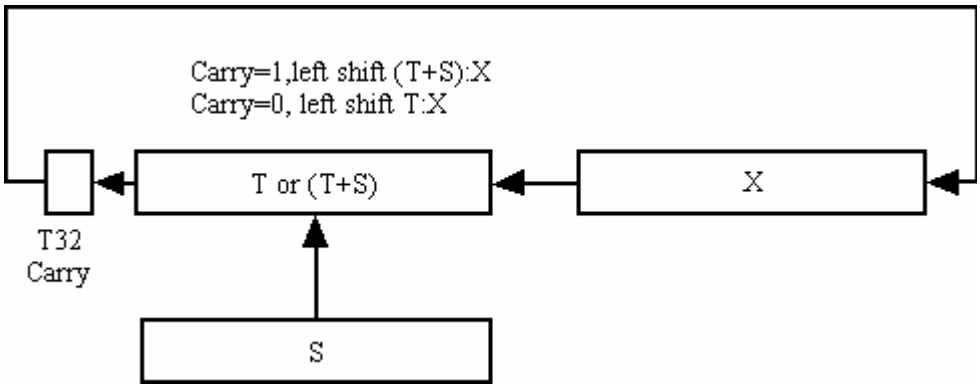

Figure 8. Division Step

3.5 Register/Stack Instructions

Instruction	Code	Function
DUP	011010	Push T on the S stack. T remains unchanged.
DROP	011111	Pop S stack to T.
NOP	011110	No operation.
OVER	011011	Push T onto S stack. Copy original contents of S to T.
POP	011000	Push T onto S stack. Pop R stack to T.
PUSH	011100	Push T onto R stack. Pop S stack to T.
TX	011101	Copy T to X. Pop S stack to T.
XT	011001	Push T onto S stack. Copy X to T.

DUP, DROP, SWAP and OVER are the 4 classic stack operations.

DUP pushes the T register on the data stack. DROP pops the data stack back into T.

SWAP exchanges T and S, the top two elements on the conceptual data stack. OVER duplicates S, and pushes it into T.

Both SWAP and OVER copy the second item on~~to~~ the stack to the top of the stack. The difference is that OVER preserves the second item in S while SWAP destroys it. We chose to implement OVER in hardware, and leave SWAP to software.

POP pops the top item on the return stack and pushes it onto the data stack. PUSH pops T from the data stack and pushes it onto the return stack. These operations are best viewed by considering return stack/R/T/S/data stack as a giant shift register array, with the three-register R/T/S window at center, exposed to the ALU. The POP instruction shifts this shift register array to the right, and the PUSH instruction shifts it to the left.

The TX and XT instructions are used to manage the X register. The X register is used to read data from memory and write data to memory. It usually holds a memory address. However, it can be used as a scratch pad register to save and restore the T register. TX pops the data stack and copies T to X. XT pushes T onto the data stack and copies the contents in X to T.

3.6 Miscellaneous Instructions

Instruction	Code	Function
EI	111110	Enable interrupts.

The eP32 provides the simplest mechanism to support real time interrupts. Five input pins on the eP32 package are allocated for real time interrupts. If interrupts are enabled, and at least one of 5 interrupt pins is not zero, a subroutine call to one of 31 locations in memory address 1 to 31 is forced on the CPU in the slot0 clock cycle. The address is selected by reading the signals on the 5 interrupt pins, and zero-extending it to form an address pointing to a memory location between 1 and 31. By filling proper branch instructions in memory locations 1 to 31 as an interrupt vector table, this microprocessor system can respond to external interrupt requests in real time.

This simple scheme allows 5 external devices to interrupt the CPU directly. If additional decoding logic were added, it could service interrupts from 31 external devices. With only 5 interrupt devices, the eP32 can respond to simultaneous interrupts from multiple devices, by constructing the interrupt vector table properly, and inserting the EI instruction properly in interrupt service routines. It is assumed that after booting, the microprocessor system configures itself so that page 0 of memory is in RAM memory, and software can change the interrupt table dynamically.

When servicing an interrupt, further interrupts are disabled and an interrupt acknowledge signal is asserted. Interrupting devices should remove their interrupt requests when seeing interrupt acknowledge. After interrupt service is completed, the interrupting service routine, or the main program must execute an EI instruction to enable future interrupts. It is a trivial matter to add a complement instruction DI to disable interrupts, but it seems to be superfluous at the moment.

Chapter 4. Implementing eP32 on the Brevia Kit

4.1 The Brevia2 Development Kit

I had opportunities to use FPGAs from Xilinx, Altera and Actel before. I implemented various versions of the eP32 on all of them. I was not particularly impressed with these companies and their FPGA products. FPGA chips were generally expensive, development boards were more expensive, and development software systems were even more expensive, bulky and usually slow.

When Lattice Semiconductor Corp announced its Brevia Development Kit at $49, I got excited. A friend Masa Kasahara loaned me his kit. I bought 2 more when Lattice had a special sale for $29. I downloaded its free development software ispLEVEL and started porting the eP32 to the LatticeXP2-5E-6TN144C FPGA chip. Working intensely for three weeks, I succeeded in getting the eP32 to work. The XP2-5E has enough logic cells to implement the eP32 CPU core, a UART, and a general purpose I/O port. It also has enough RAM memory to host the eForth operating system. The nicest thing is that its RAM memory is mirrored in on-chip flash memory, and the entire eP32 system is contained in a single XP2-5E chip. All other FPGAs required external components to host a complete microprocessor system. The XP2 is my dreamed SOC chip.

My only complaint is that its software development system, ispLEVER, is too bulky. It required me to free up 5 GB of disk space to install it, with accompanying Synplicity synthesis tools and Aldec ActiveHDL simulation tools. One other thing is that the Brevia Kit requires a COM port and a parallel printer port on my PC for communication and for a JTAG interface. It is not a big deal for me, because I have this old desktop computer, which has these ports.

Recently Lattice replaced the Brevia Kit with Brevia2 Kit, and upgraded ispLEVEL to Diamond IDE. Two cables connecting to the COM and printer ports were replaced by a single USB cable.. The eP16r implementation is tested and verified on the Brevia2 Kit, with Diamond 1.4 IDE system. I had trouble installing the USB drivers on on of my PC, but that's another story.

Here is a laundry list of components included in the Brevia Kit:

- LatticeXP2 FPGA: LFXP2-5E-6TN144C
- 2 Mbit SPI Flash Memory
- 1 Mbit SRAM
- A single USB cable for programming and communication
- 2x20 and 2x5 Expansion Headers
- Push buttons for General Purpose I/O and Reset
- 4-bit DIP Switch for user-defined inputs
- 8 Status LEDs for user-defined outputs

Since the XP2-5E has 166K bits of embedded block RAM, I do not need the external SPI flash memory and SRAM. The USB interface actually implemented two devices: an UART port for communication, and a parallel port to program the FPGA. The LEDs, push buttons, and switches are very useful for demonstrations. This kit

has everything I need to demonstrate my eP32 microprocessor design and the eForth operating system.

Here I will show you steps to get the eP32 implemented on my Brevia2 Kit and to get the eForth system to run, talking to HyperTerminal on a PC.

You have to download the Diamond IDE suite from www.latticsemi.com to implement the eP32. You need the Diamond System for Windows, the Synplify Synthesis Module, and the Aldec Active-HDL Lattice Web Edition Module. They take up a huge amount of disk space. Then you have to apply for a license from Lattice. Lattice also provides many examples for you to evaluate. You may want to look at their Demo Application, which contains a LatticeMico8 Reference Design. LatticeMico8 is an 8-bit microprocessor. Only after you studied LatticeMico8 will you appreciate that the eP32, a 32-bit microprocessor, can be simpler than an 8-bit microprocessor with conventional architecture.

4.2 Synthesize the eP32

You have to install Diamond first. When Diamond is up and running, open a new project. Name this project eP32, if you do not have a better name. A New Project Wizard will help you set up this project. You have to select LatticeXP2-5E as your target device and VHDL as your programming language. Now, import the following files into the above project.

File	Module
ep32_chip.vhd	Top level microprocessor system
ep32.vhd	eP32 CPU module
ram_memory.vhd	RAM memory module
uart.vhd	Serial UART module
gpio.vhd	General purpose parallel IO module
ep32q_tb.vhd	Test bench for the eP32 system

In the Diamond Project panel, select the File List tab. You will see that all the above files are imported as shown in Figure 9.

Click the Process tab in the Project panel, and you will see the modules arranged in a hierarchy as in Figure 10:

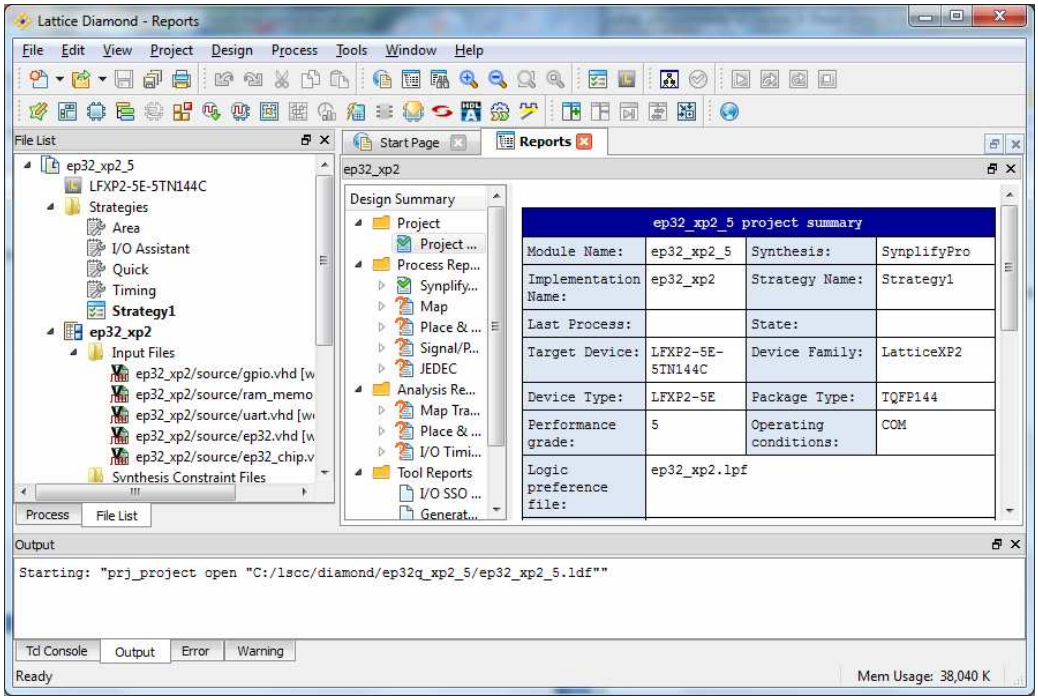

Figure 9. Diamond IDE, File List

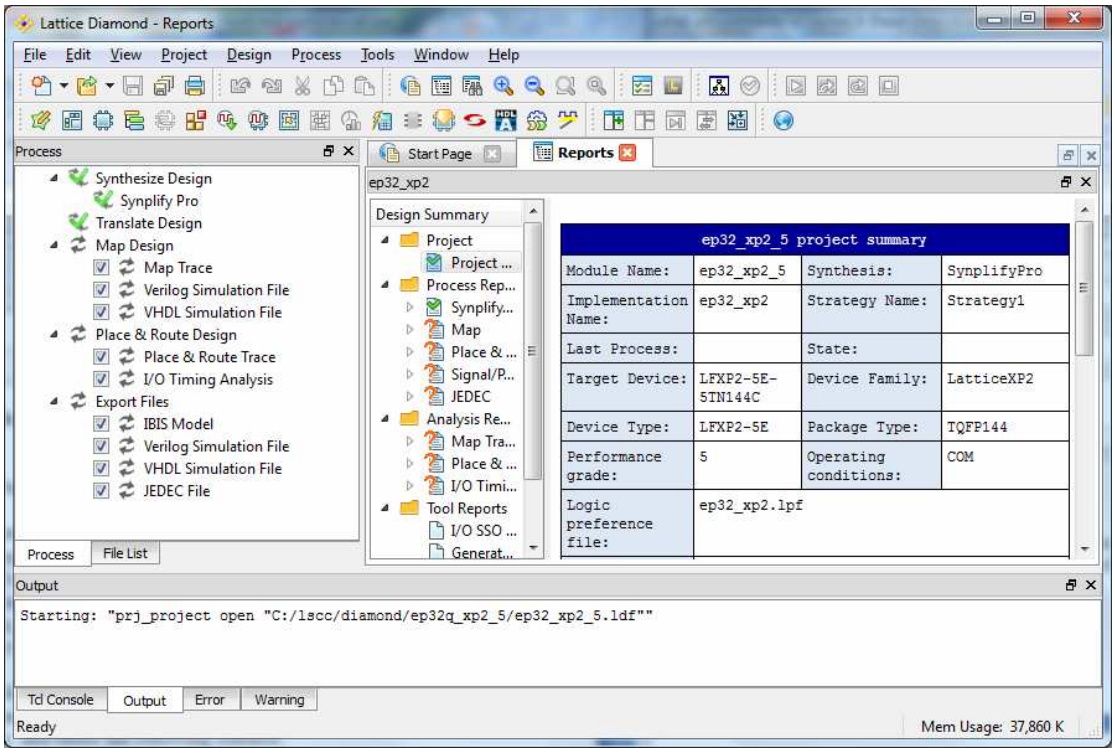

Figure 10. Diamond IDE, Process View

As I ported the eP32 design from a project using an Altera FPGA, ep32.vhd, uart.vhd, and gpio.vhd all remain unchanged and Syplicity compiles them correctly. ram_memory.vhd was changed to use the RAM_DQ module provided in the Diamond system. If you change the eForth system and get a new target image in mem.mif, you have to generate a new ram_memory.vhd file, so that the new eForth target image

can be included in ram_memory.vhd.

To change ram_memory.vhd, click Tools>IPexpress to invoke IPexpress. Select RAM_DQ module. Fill in a file name of ram_memory and select VHDL as module output, and you get a screen like Figure 11.:

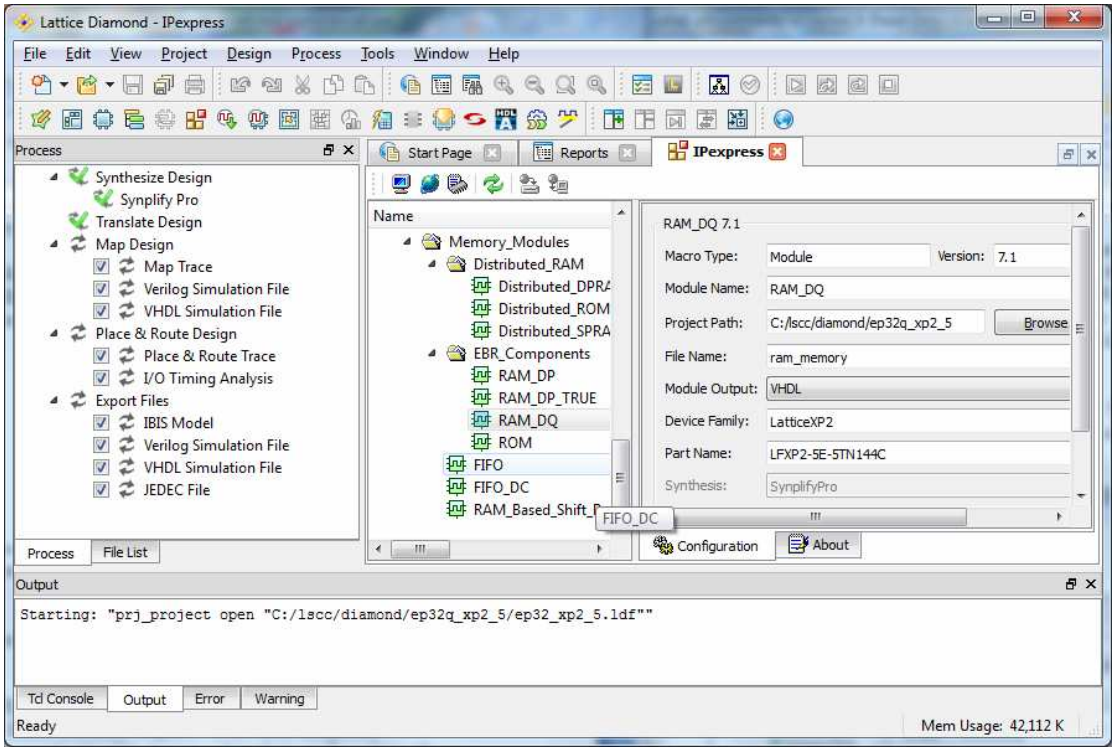

Figure 11. RAM_DQ in IExpress

Click the Customize button, and you get a RAM_DQ configuration panel, like that shown in Figure 12. Make the following selections:

Memory depth: 4096
Memory width: 32 bits
No output latch
Memory type: synchronous
Optimization: time
Initializing file: mem.mif
File type: Hex-address

Click the Generate button and a new mem_memory is produced. There is a ram_memory_templ.vhd file containing the VHDL configuration code you can copy and paste into ep32_chip.vhd.

In the Project panel, click Process tab and select all the process boxes, as shown in Figure 13.

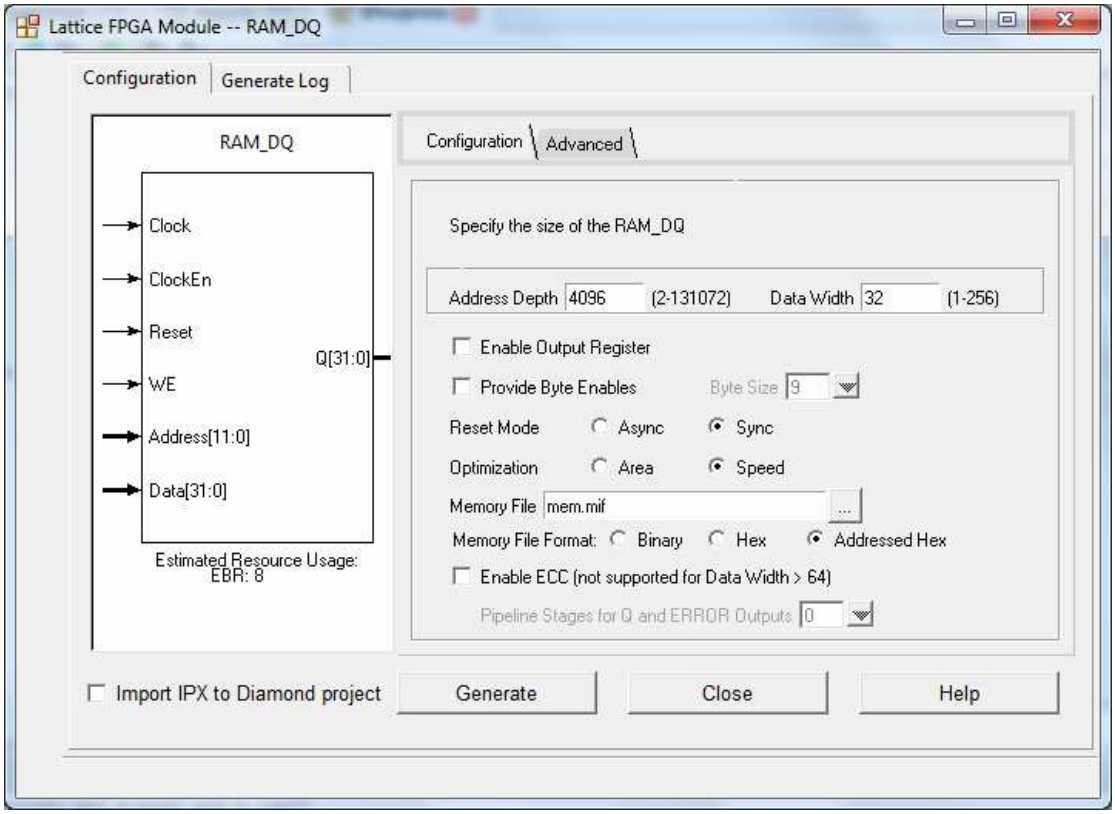

Figure 12. RAM_DQ Module Configuration

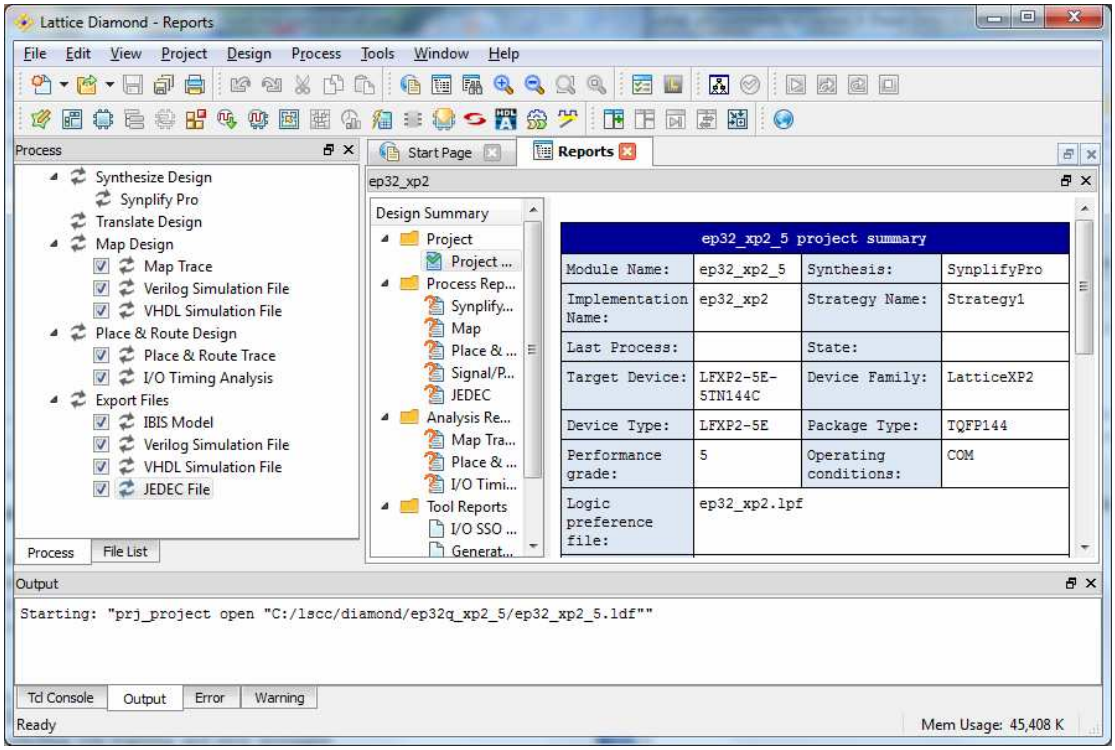

Figure 13. Select Synthesis Process

Pull down the Process Menu and select the Rerun All button. It invokes Synplicity Synthesis tools to analyze and to synthesize this design. Synplicity will analyze all

VHDL files and synthesize this design accordingly. After each process step, a green check mark is places after each selection box to indicate that this step is completed successfully.

If you are to change this design, this is probably the place you will spend lots of time editing and adding to your VHDL files and then run Synplicity Synthesizer. The synthesizer is very generous in sending you warning and error messages. Look up each error message and try to fix the problem in your VHDL files.

4.3 Simulate the eP32

Lattice bundles Active-HDL simulation tools from Aldec in the Diamond system. Active-HDL itself is a very complicated system, and you need to spend considerable time learning it.

In the older ispLEVEL IDE, you need a test bench VHDL file to simulate your design. It can generate a template of a test bench for any VHDL module in your design, to help you build the test bench. In Diamond, you can specify simulation functions to input signals directly, and a test bench file is not needed.

Pull down the Tools Menu and select the Simulation Wizard button. The Active-HDL simulator starts and shows you a series of windows. One window asks you for a project name. Another asks you to confirm your RTL simulation level. Just click the Next> button until the simulator is actually loaded. Then you get a screen like Figure 14.

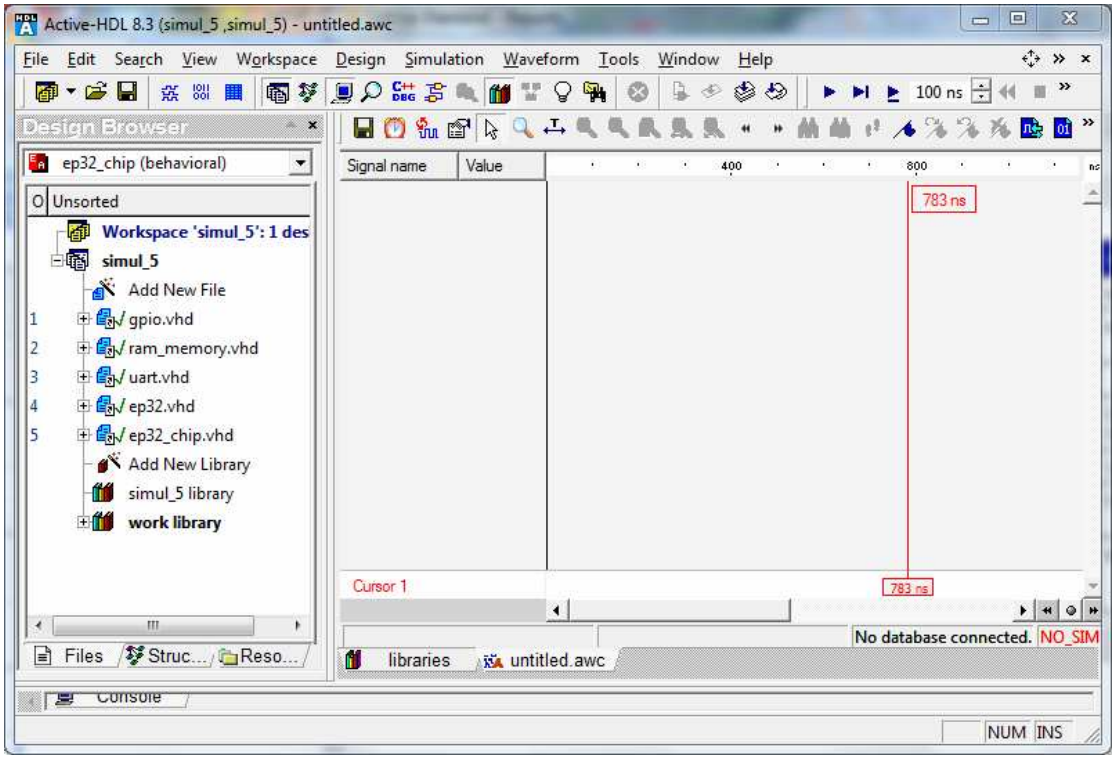

Figure 14. HDL Simulator

On the Design Browser panel to the left, click the Structure tab at the bottom, then

select the eP32_chip(Behavioral) model button, and you get a list of signals as shown in Figure 15.

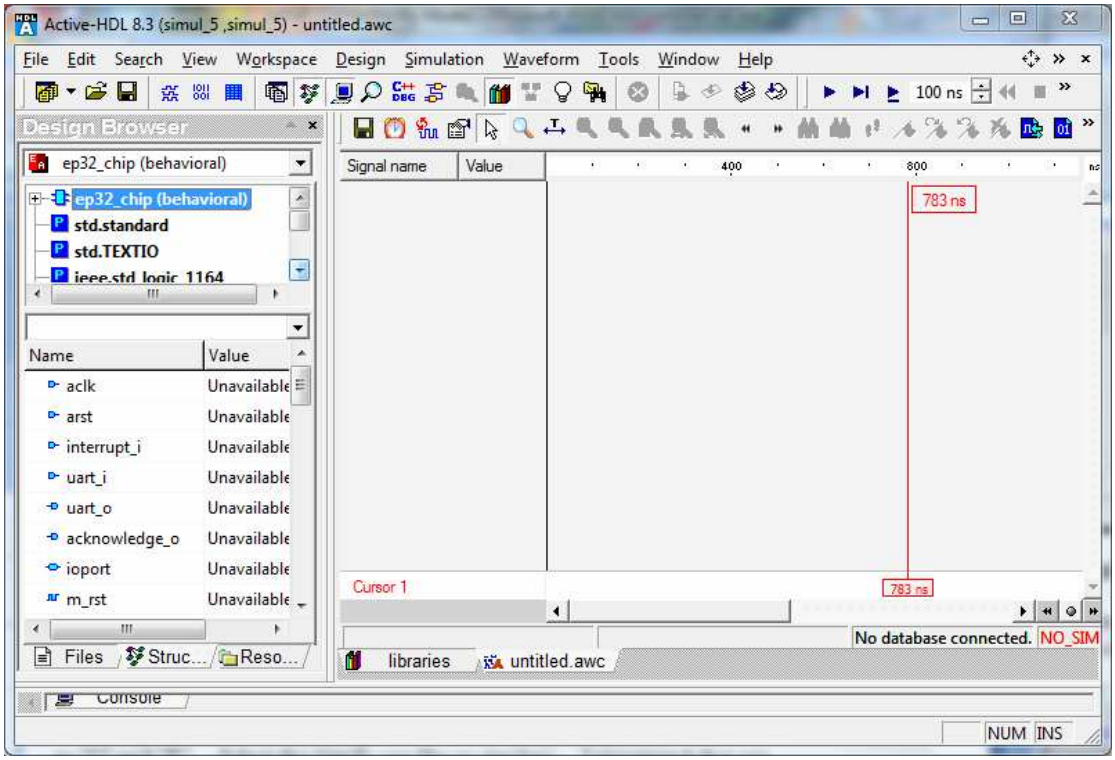

Figure 15. Select cp32 chip module

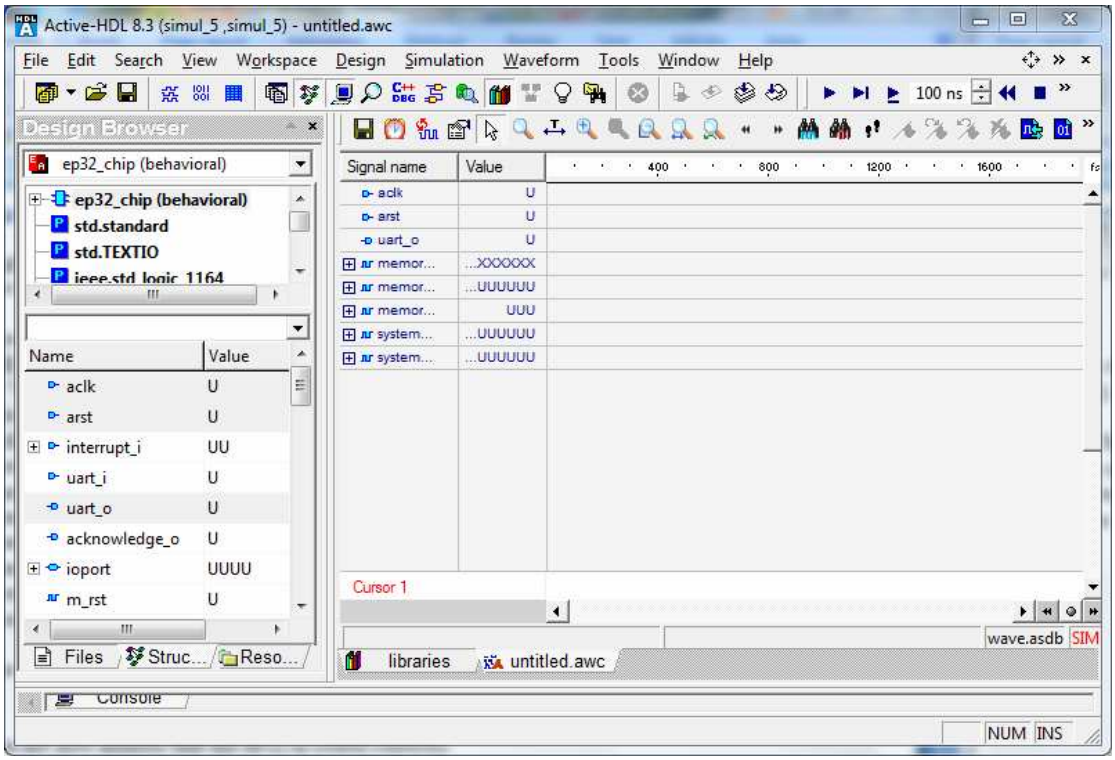

Figure 16. Select Simulation Signals.

Now, pull down the Simulation Menu and select the Initialize Simulation button. It will change the values of all the signals in the eP32_chip design from "Unavailable" to "U" and "X". Select the signals you like to simulate. I reccomand that you select the following signals:

Aclk
Arst
Uart_o
Memory_data_o
Memory_data_i
Memory_addr
System_addr
System_data_o

Right chick on the selected signals and select "Add to Waveform" option and you will see the screen as shown in Figure 16.

Before running the simulation, you have to specify two input signals aclk and arst. Right click the aclk under "Signal Name" and select the "Simulators…" option in the pop-up menu. The Simulators window pops up. Select "Clock" in the "Type" panel, and you get the screen shown in Figure 17.

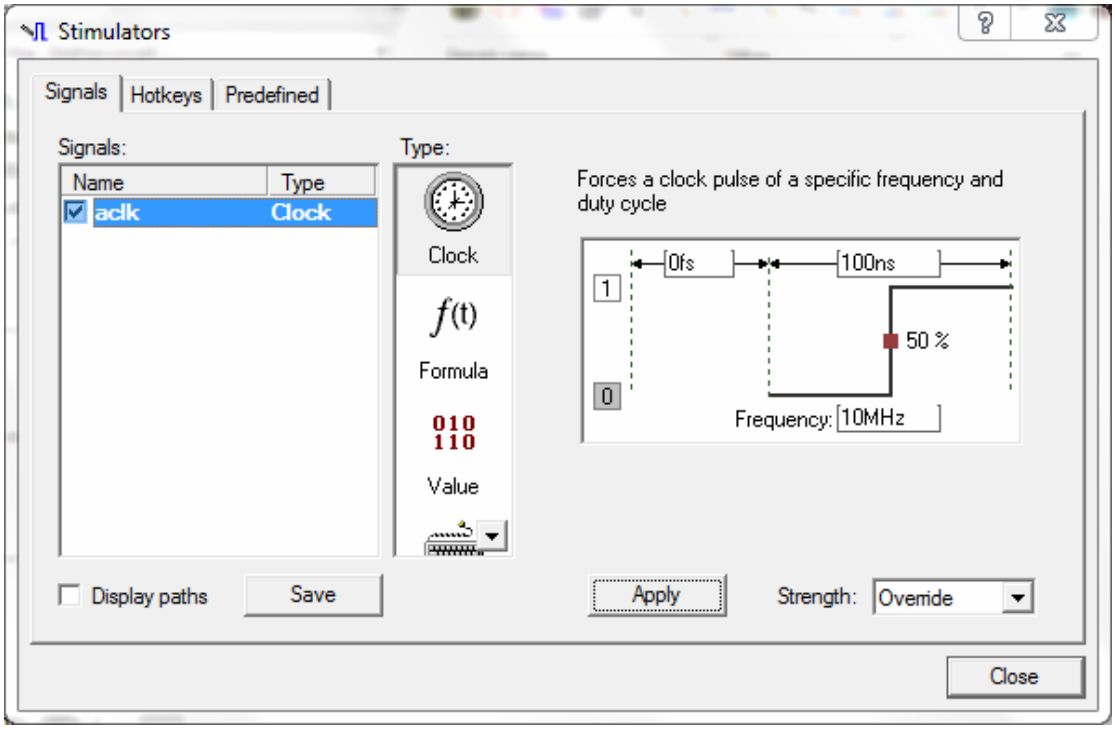

Figure 17. Simulate Master Clock

Click the Apply button and then the Close button to confirm that you apply a 10 MHz clock signal to aclk input.

Right click the arst signal under "Signal Name" and select the "Simulators…" option in the pop-up menu. The Simulators window pops up. Select "Formula" in the "Type" panel, and specify that the reset signal starts at "0" level for 1000 ns and then

changes to “1”. Now you get the the screen shown in Figure 18.

Click the Apply button and then the Close button to confirm that you apply the proper reset signal to arst input.

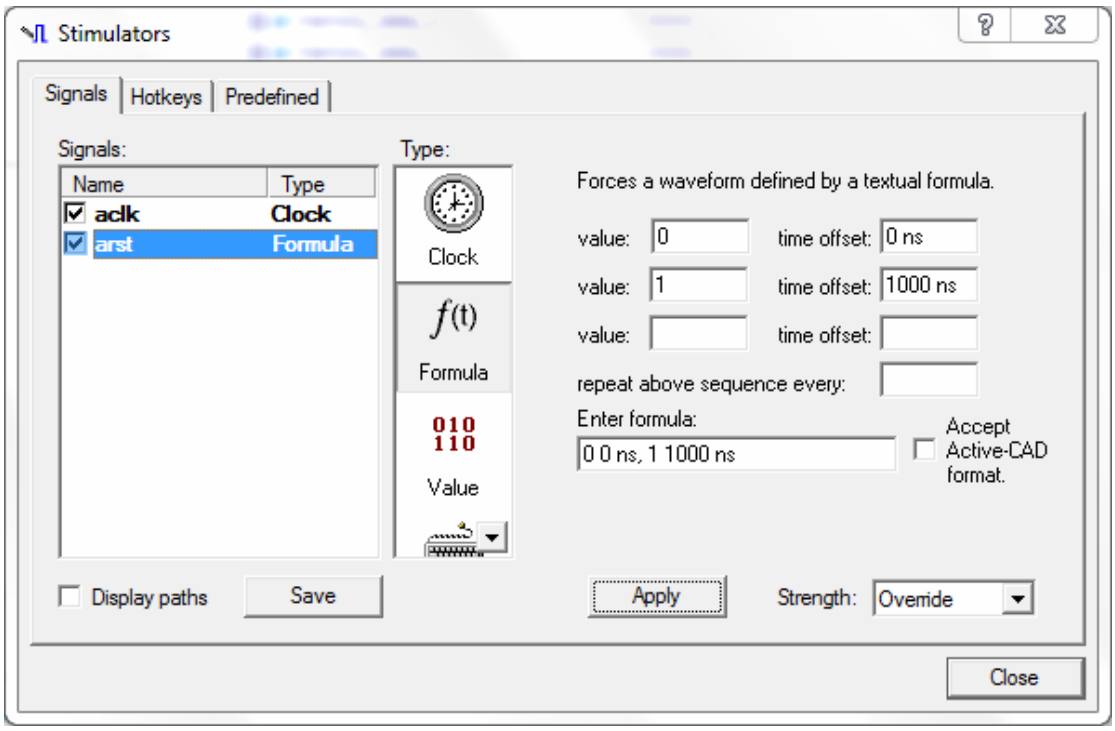

Figure 18. Simulate Master Reset

Now, pull down the Simulation Menu and select the Run Until button. Enter “1 ms” in the data box to let the simulator run for 1 ms:

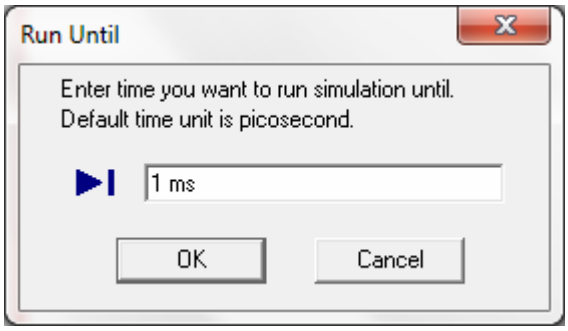

Figure 19. Select Simulation Time

Click the OK button and the simulator produces the waveforms as shown in Figure 20.

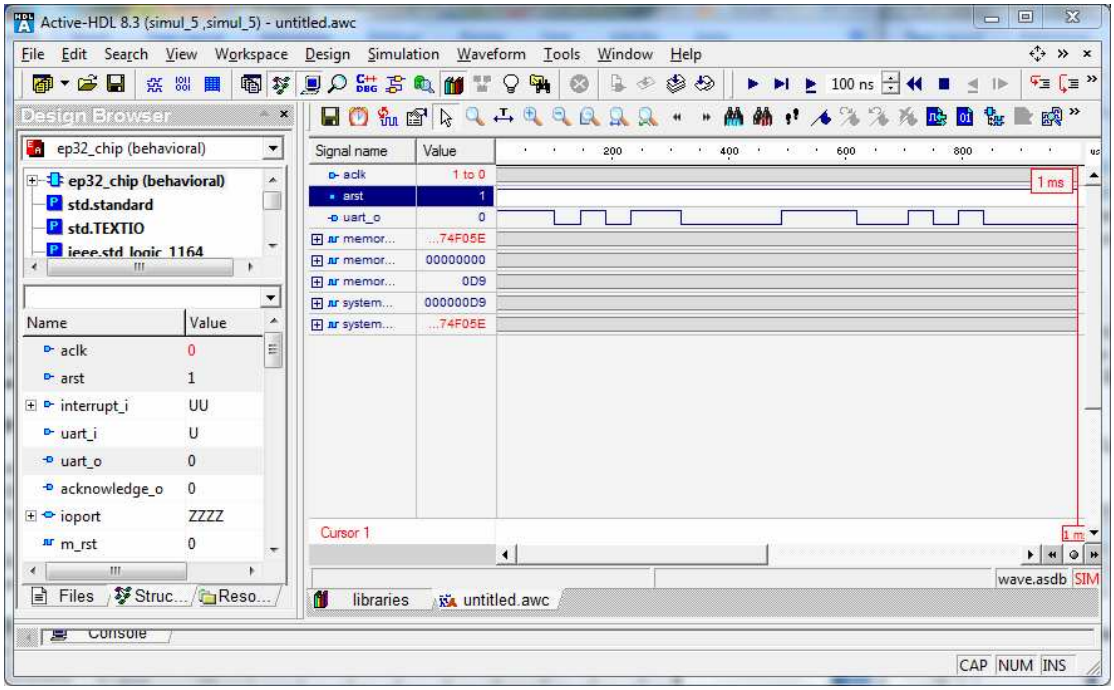

Figure 20. Simulation Waveforms

Look at the signal uart_o. It is showing that Ep32 sends out a Carriage Return (ASCII 0xD) and a Line Feed (ASCII 0xA) character. You are now assured that the eP32 is coded correctly.

Click the Zoom In button (A magnifier glass with a + sign) 12 times, and drag the waveforms to the beginning to the left, you will see this screen in Figure 21.

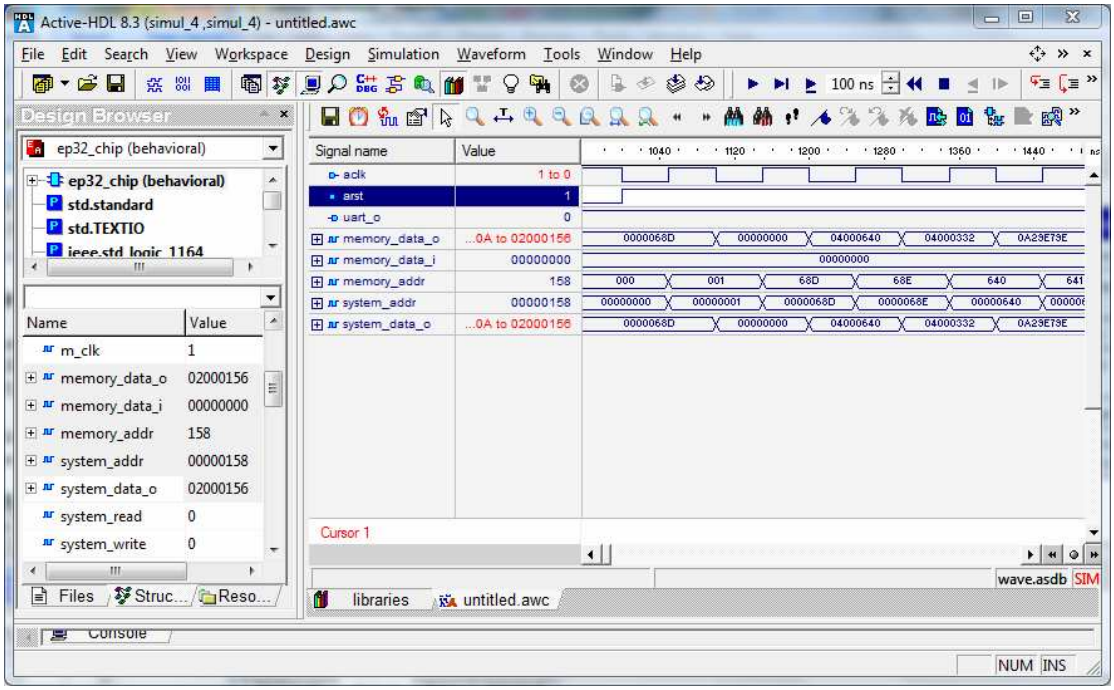

Figure 21. Expanded View of the Waveforms

The signals memory_addr and system_addr make the following sequence of changes:

```
000->001->68D->68E->640->641
```

which show that eP32 starts at address 0 on reset, jumps to COLD, which calls DIAGNOSE. These are the correct sequence of instructions after eP32 starts. You are now completely assured that the eP32 is running correctly.

4.4 Layout the eP32

After logic design of the eP32 is verified by synthesis and simulation, you have to assign input and output signals to proper pins on the XP2-5E-5TN144C chip according to the board layout of the Brevia Kit, so that you can actually run the eP32 on the Brevia2 Kit.

Pull down Tools Menu, and select Package View. In Package View, you see a Package panel on the right in Figure 22.

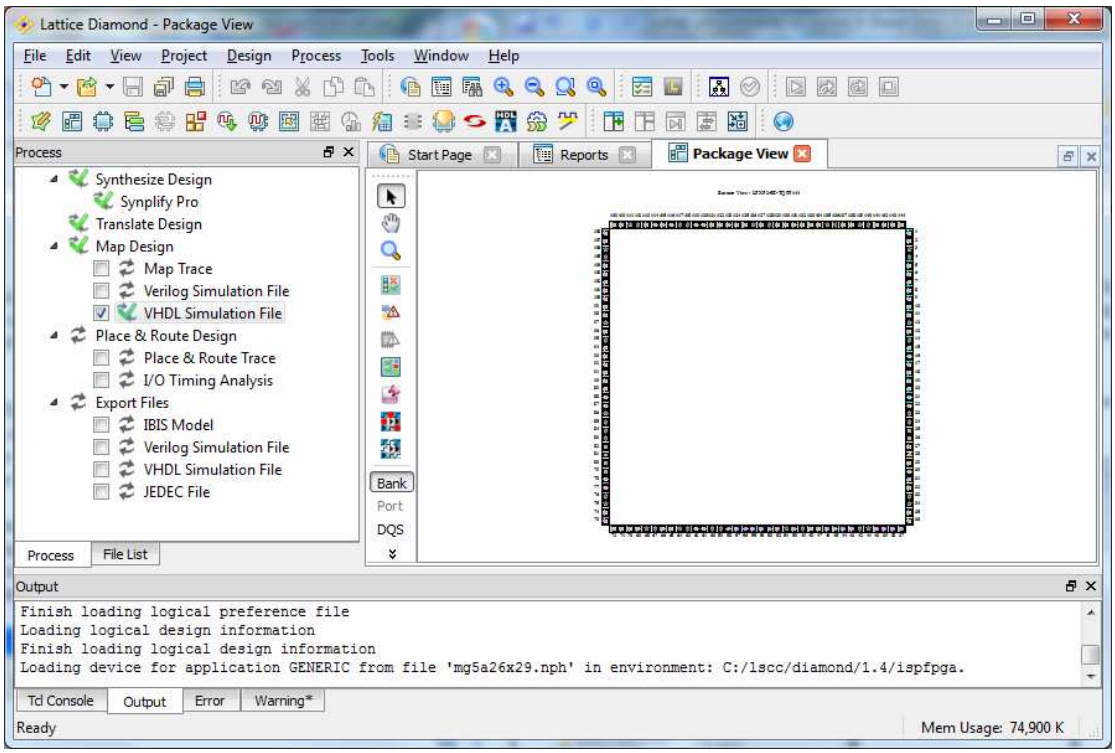

Figure 22. Package View of XP2 Chip

Pull down the View Menu and select Preference Preview, you get to see the contents of the preference file eP32.pdf. It looks like that in Figure 23.

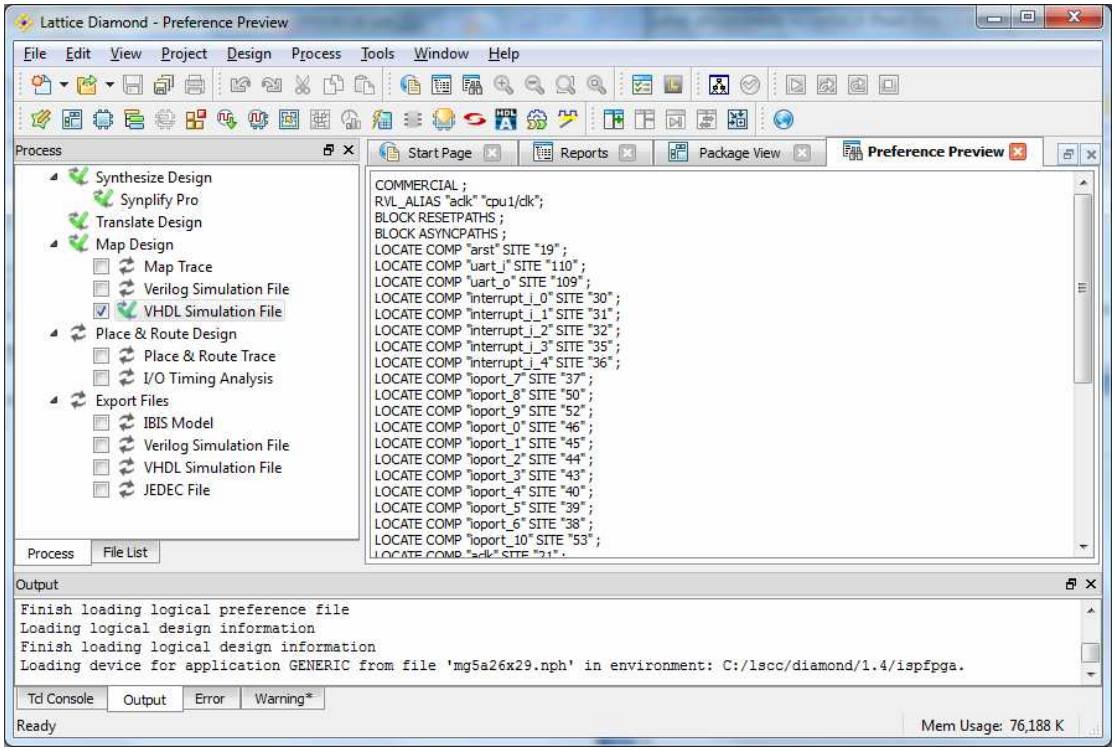

Figure 23. Pin Assignments of eP32

Signals on the eP32 chip and their corresponding pins on the XP2-5E-5TN144C chip package are listed in the following table:

Signal	Pin Number
aclk	21
arst	19
interrupt_i[0]	58
interrupt_i[1]	57
interrupt_i[2]	56
interrupt_i[3]	55
interrupt_i[4]	54
ioport[7]	37
ioport[8]	53
ioport[9]	52
ioport[0]	46
ioport[1]	45
ioport[2]	44
ioport[3]	43
ioport[4]	40
ioport[5]	39
ioport[6]	38
ioport[10]	50
ioport[11]	1
ioport[12]	2
ioport[13]	5

ioport[14]	6
ioport[15]	7
uart_i	110
uart_o	109

You have to get the signals assigned to correct pins; otherwise, the eP32 will not work on the Brevia2 Kit. Other minor things like clock frequency and signal delays do not affect the implementation, except that you will get lots of warning messages complaining that physical layout does not meet timing and delay requirements.

4.5 Programming eP32

The Brevia2 Kit includes a USB cable to connect to a PC. Connect Brevia2 Kit to your PC. If you have done the systhesis and simulation of eP32 correctly, you can now program eP32 to Brevia2 and test eP32.

Bring up Diamond, and open the eP32 project. Pull down Tools Menu and select Programmer. A Programmer window opens up like that shown in Figure24.

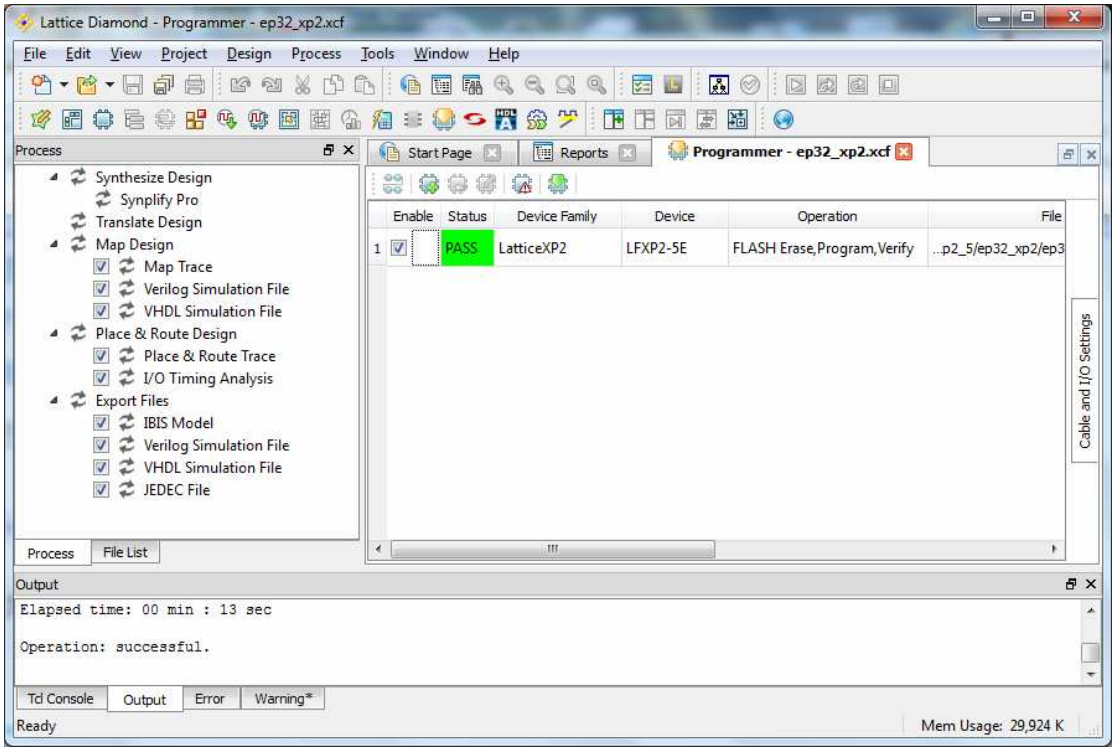

Figure 24. Diamond Programmer

From the File Name section, click the Browse button. The File Name window appears. Browse to the eP16 project folder, select the ep32_xp2.jed file, and click the Open button. From the Operation list, choose Flash Erase, Program, Verify, and click the OK button.

The last button to the right on the top of the Programmer Panel is the Program button. Click it and Diamond reprograms the XP2 chip on the Brevia2 Kit.

If the UART cable is connected to a COM port on the PC, and HyperTerminal is already opened and configured to 115,200 baud, 1 start bit, 8 data bits, 1 stop bit, no parity, and no flow control, you should see that the eP32 boots up and displays a sign-on message, “eP32q v2.05”, as shown in Figure 25:

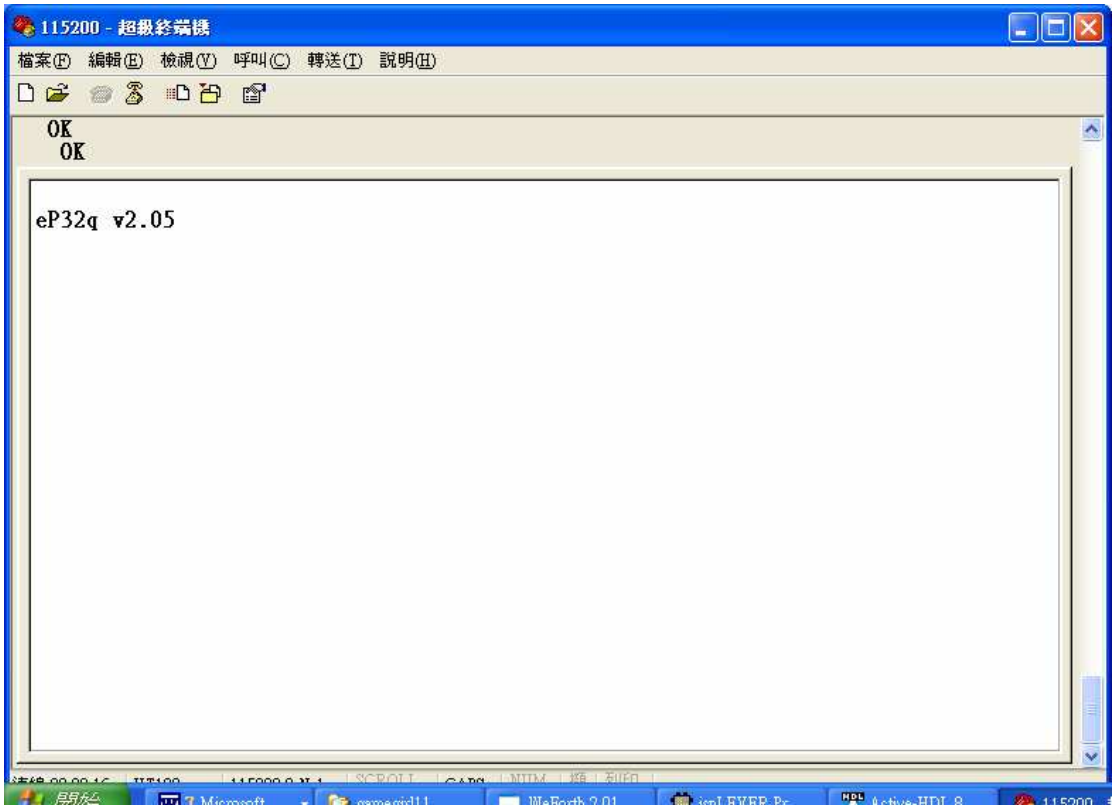

Figure 25. eP32 Sign-on Message

You can now type in FORTH commands and interact with the eForth system that runs on the eP32 microprocessor you just downloaded to the Brevia Board.

Type these commands:
: TEST1 CR ." HELLO, WORLD!" ;
TEST1

You will see that eForth produces the results as shown in Figure 26:

To demonstrate that you have full control over the Brevia Board, let us do some exercises on the GPIO port. First, here are the registers in the GPIO module, which we can access by reading and writing to memory locations 0xE0000000-0xE0000002:

Address	Register	Function
0xE0000000	gpio_out	When written, send data to gpio port
0xE0000001	gpio_dir_reg	Select port pin direction: 0-input; 1-output
0xE0000002	gpio_in	Read gpio port

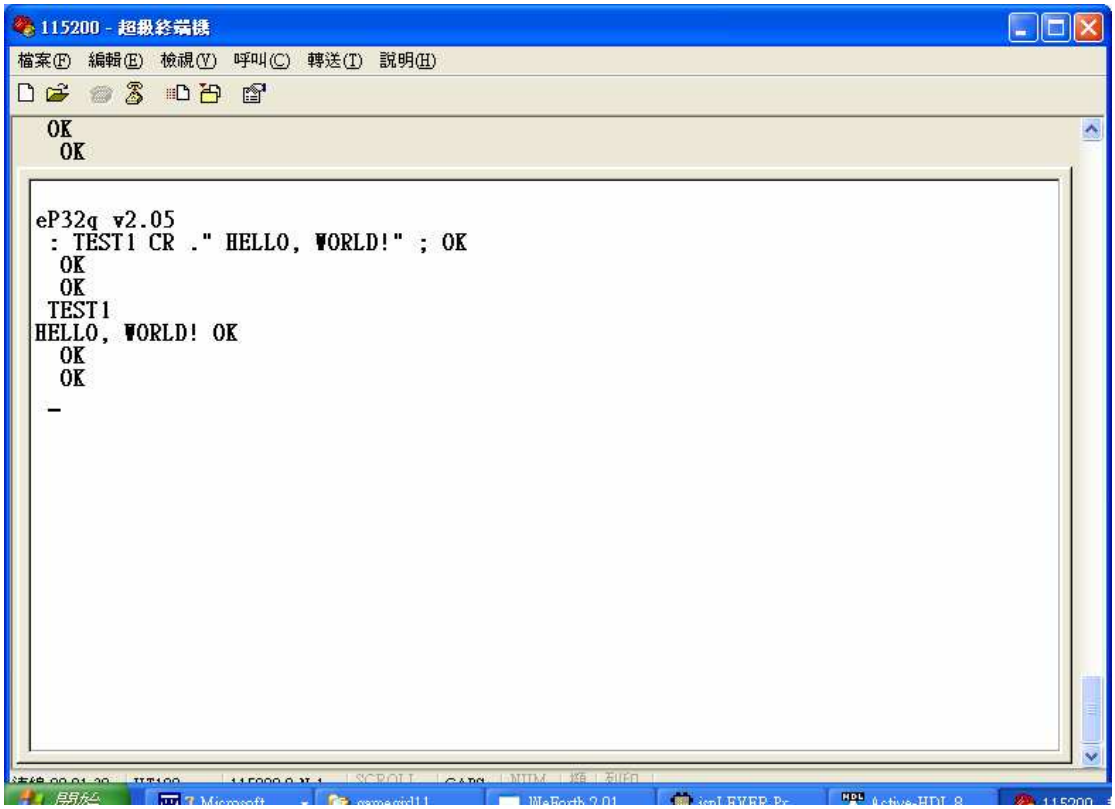

Figure 26. The Universal Greeting

Type the following commands to configure the lower 8 bits in the GPIO port as outputs and the next upper 8 bits as inputs:

HEX
FF E0000001 !

Now, you will see that all 8 LED's on the Brevia are turned on. To turn them off, type:
FF E0000000 !

To turn on only one LED, type:
FE E000000 !

To read the push button switches on the Brevia Board, type:
E0000002 ?

FFFE is the result displayed. The lower 8 bits (FE) show that only one LED was turned on. The upper 8 bits (FF) show that all push button switches are off. Push down switch SW5 and type:
E0000002 ?

The returned results change to FDFE, as closing SW5 pulls down bit 9 of the GPIO port.

The above exercises leave this display on HyperTerminal, as shown in Figure 27:

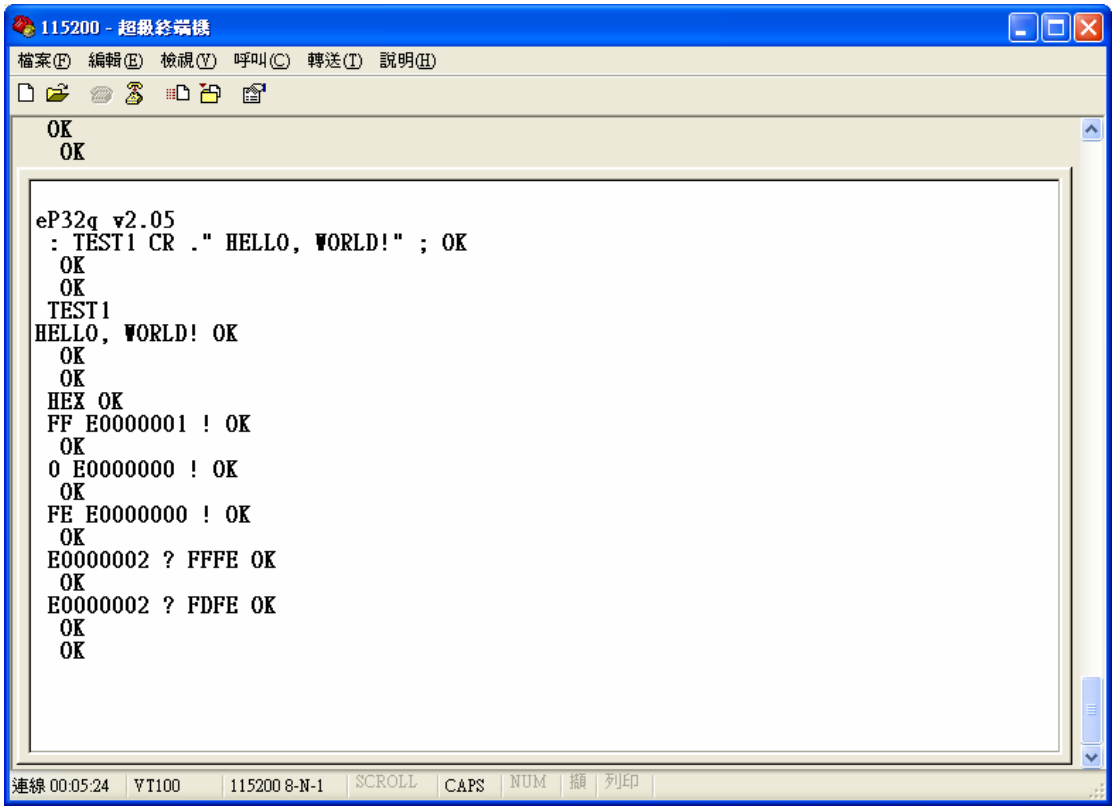

Figure 27. IO Exercises on Brevia2 Kit

These exercises should be very convincing that you have a nice interactive operating system hosted on the top of a very versatile and powerful 32-bit microprocessor. All these things on a $49 FPGA development kit!

Chapter 5. The eP32 Design in VHDL

Here I will describe a complete 32-bit microprocessor designed in VHDL. It includes a CPU core, a RAM memory module, a UART, and a general purpose GPIO port. Together with the eForth operating system produced by a metacompiler, I build a complete running Forth system, ready for application development. It is a complete hardware and software development system to explore SOC applications. The FPGA chip LatticeXP2-5E can host this complete microprocessor system, and it is implemented on the LatticeXP2-5E Brevia Development Kit, using the ispLEVER FPGA Development Software System..

In the following sections, I will present VHDL code in the following files implementing various modules of the eP21 microprocessor system:

File	Module
ep32_chip.vhd	Top level microprocessor system
ep32.vhd	eP32 CPU module
ram_memory.vhd	RAM memory module
uart.vhd	Serial UART module
gpio.vhd	General purpose parallel IO module

Following is a block diagram of the eP32 chip, showing modules in it and signals and busses connecting these modules:

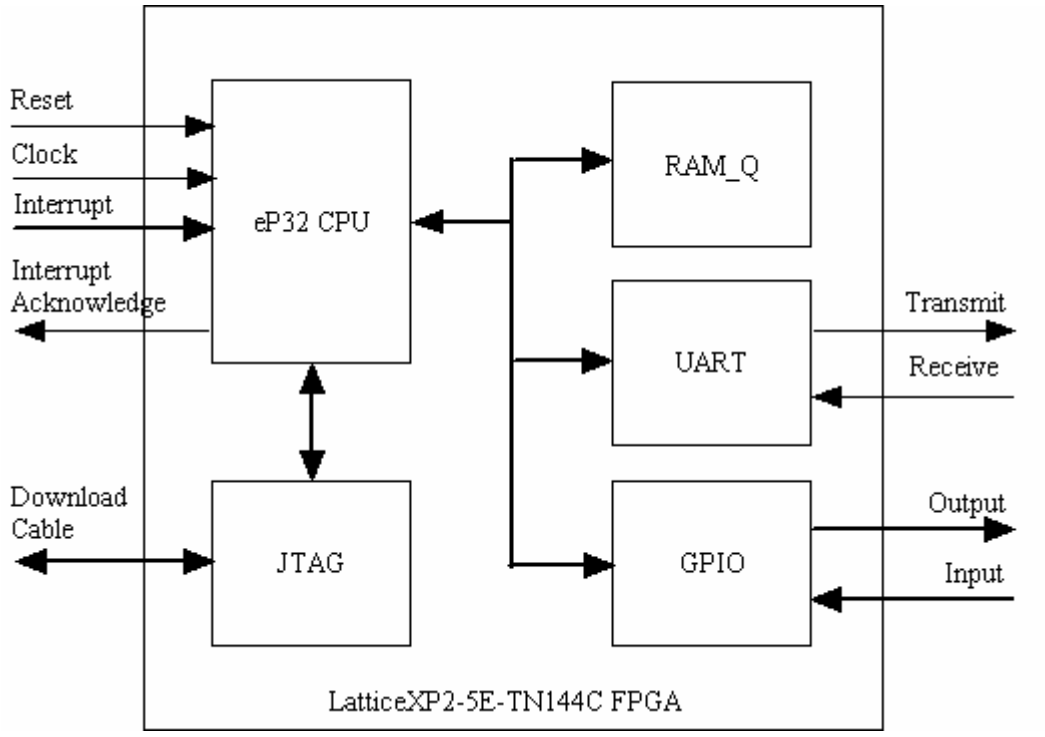

Figure 28. Components in eP32 Chip

5.1 Top Level eP32 Chip

VHDL code in ep32_chip.vhd instantiates all modules in the eP32 system.

Here are port signals defined for the top level eP32 chip. Since RAM is implemented as an internal module, it is not necessary to bring out address and data signals from the CPU core to the chip package. Therefore, only aclk, arst, interrupt_i, acknowledge_o, uart_i, uart_o and useful GPIO pins are necessary to implement a chip that runs the eForth system for program development. This eP32 system can be hosted in a very small package with 8-14 pins.

I/O pins of this eP32 chip and their functions are as follows:

Port Signal	Function
aclk	External clock input
arst	External reset input
interrupt_i	External interrupt input
acknowledge_o	Interrupt acknowledge
uart_i	UART receiver input
uart_o	UART transmitter output
ioport	General purpose I/O port

In component declarations, the following modules are declared:

Component Module	Function
ep32	eP32 CPU core module
ram_memory	RAM memory module
uart	Serial UART module
gpio	General purpose parallel I/O module

These modules are later instantiated and all their ports are connected to signals defined in the top level system module.

eP32 Module

The eP32 module is a complete CPU core. Its input/output signals are as follows:

clk	Input master clock
clr	Input master reset
interrupt	Input external interrupt
data_i	Input data bus
intack	Output interrupt acknowledge
read	Output memory/io read enable
write	Output memory/io write enable
addr	Output address bus
data_o	Output data bus

```
--
************************************************************
-- *       (C) Copyright 2002, eForth Technology Inc.          *
-- *                   ALL RIGHTS RESERVED                    *
--
*============================================================*
-- * Project:           FG in PROASIC                          *
-- * File:             ep32_chip.vhd                          *
-- * Author:            Chien-Chia Wu                          *
-- * Description:        Top level block                        *
-- *                                                          *
-- * Hierarchy:parent:                                          *
-- *          child :                                          *
-- *                                                          *
-- * Revision History:                                          *
-- * Date        By Who        Modification                    *
-- * 09/19/02    Chien-Chia Wu   Branch from ep16a.              *
-- * 01/02/03    Chien-Chia Wu   Add SDI.                        *
-- * 01/29/03    Chien-Chia Wu   Add Boot.                       *
-- * 02/24/03    Chien-Chia Wu   Modify the module as 32-bits    *
-- *                           version.                       *
-- * 02/27/03    Chien-Chia Wu   Modify SDRAM as byte-assecable. *
-- * 03/02/03    Chien-Chia Wu   Add internal SRAM module.      *
-- * 06/29/06    Chen-Hanson Ting Add HMPP/Shifter/Controller.   *
-- * 11/18/10    Chen-Hanson Ting Port to LatticeXP2 Brevia Kit  *
--
************************************************************

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_misc.all;
use ieee.std_logic_unsigned.all;

entity ep32_chip is
port(
    -- input port
    aclk:             in     std_logic;
    arst:             in     std_logic;
    interrupt_i:       in    std_logic_vector(4 downto 0);
    -- input port
    uart_i:          in     std_logic;
    -- output port
    uart_o:          out    std_logic;
 -- GPIO Interface
    ioport:          inout  std_logic_vector(15 downto 0)
  );
end entity ep32_chip;
```

UART module

The UART module conforms to standard RS232 UART specifications, although we only use two I/O pins, rxd_i and txd_o. No handshake or flow control signals are used. Input/output signals in the UART module are as follows:

rst_i	Input reset
ce_i	Input chip enable
read_i	Input read enable
write_i	Input write enable
addr_i	Input address bus
data_i	Input data bus
data_o	Output data bus
rx_empty_o	Output receiver empty flag
rx_irq_o	Output receiver interrupt request
tx_irq_o	Output transmitter interrupt request
rxd_i	Input receiver data
txd_o	Output transmitter data
cts_i	Input clear to send
rts_o	Output ready to send

RAM Module

The RAM_MEMORY module is configured to use the RAM_Q memory of embedded block memory EBR in the LatticeXP2-5E FPGA chip. Input/output signals are as follows:

Clock	Input master clock
ClockEn	Input clock enable
Reset	Input master reset
WE	Input write enable
Address	Input address bus
Data	Input data bus
Q	Output data bus

GPIO Module

Input/output signals are as follows:

clr	Input master reset
clk	Input master clock
write	Input write enable
read	Input read enable
ce	Input chip enable
addr	Input address bus
data_in	Input data bus
gpio_in	Input GPIO data
data_out	Output data bus
gpio_out	Output GPIO data
gpio_dir	OutPut GPIO direction

```
architecture behavioral of ep32_chip is
  -- component declaration
  component ep32 is
    port(
      -- input port
      clk:       in     std_logic;
      clr:       in     std_logic;
      interrupt:  in     std_logic_vector(4 downto 0);
      data_i:    in     std_logic_vector(31 downto 0);
      intack:    out    std_logic;
      read:      out    std_logic;
      write:     out    std_logic;
      addr:      out    std_logic_vector(31 downto 0);
    data_o:    out    std_logic_vector(31 downto 0)
    );
  end component;

  component uart is
    port(
      -- input
      clk_i:        in     std_logic;
      rst_i:        in     std_logic;
      ce_i:         in     std_logic;
      read_i:       in     std_logic;
      write_i:      in     std_logic;
      addr_i:       in     std_logic_vector(1 downto 0);
      data_i:       in     std_logic_vector(31 downto 0);
      -- output
      data_o:       out    std_logic_vector(31 downto 0);
      rx_empty_o:   out    std_logic;
      rx_irq_o:     out    std_logic;
      tx_irq_o: out    std_logic;
      -- external interface
      rxd_i:        in     std_logic;
      txd_o:        out    std_logic;
      cts_i:        in     std_logic;
      rts_o:        out    std_logic
    );
  end component;


component ram_memory
    port (Clock: in  std_logic; ClockEn: in  std_logic;
       Reset: in  std_logic; WE: in  std_logic;
       Address: in  std_logic_vector(11 downto 0);
       Data: in  std_logic_vector(31 downto 0);
       Q: out  std_logic_vector(31 downto 0));
end component;
```

Top Level Global Signals

Here are global signals defined in the top level eP32 chip. Their principal purposes are connecting port signals of instantiated modules. However, many signals are defined in terms of logical equations constructed from other signals. These logical equations are then presented with relevant modules.

The following are Global signals in the eP32 chip:

Signal	Function
m_rst	Inverted master reset
m_clk	Inverted master clock
memory_data_o	Memory data output bus
memory_data_i	Memory data input bus
memory_addr	Memory address bus
system_addr	System address bus
system_data_o	System data output bus
system_read	System read enable
system_write	System write enable
system_ack	system interrupt acknowledge
cpu_data_i	CPU data input bus
cpu_addr_o	CPU address bus
cpu_data_o	CPU data output but
cpu_m_read	CPU memory read enable
cpu_m_write	CPU memory write enable
cpu_intack	CPU interrupt acknowledge
cpu_ready_i	CPU ready input
cpu_ack_o	CPU interrupt acknowledge output
uart_ce	UART chip enable
uart_addr	UART address bus
uart_data_i	UART data input bus
uart_data_o	UART data output bus
uart_rx_empty	UART receiver empty flag
uart_rx_irq	UART receiver interrupt request flag
uart_tx_irq	UART transmitter interrupt reuqest flag
uart_rxd	UART receiver data
uart_txd	UART transmitter data
uart_cts	UART clear to send
uart_rts	UART ready to send
gpio_ce	GPIO chip enable
gpio_addr	GPIO address bus
gpio_data_i	GPIO data input bus
gpio_in	GPIO input pins
gpio_data_o	GPIO data output bus
gpio_out	GPIO output pins
gpio_dir	GPIO input/output direction

```
component gpio
  port(
    -- input port
    clr: in         std_logic;
    clk: in         std_logic;
    write: in       std_logic;
    read: in        std_logic;
    ce: in            std_logic;
    addr: in           std_logic_vector(1 downto 0);
    data_in: in        std_logic_vector(31 downto 0);
    gpio_in: in       std_logic_vector(15 downto 0);
    -- output port
    data_out: out      std_logic_vector(31 downto 0);
    gpio_out: out     std_logic_vector(15 downto 0);
    gpio_dir: out     std_logic_vector(15 downto 0)
  );
end component;

  -- interal globle signal
  signal m_rst:              std_logic;
  signal m_clk:              std_logic;
  signal memory_data_o:       std_logic_vector(31 downto 0);
  signal memory_data_i:       std_logic_vector(31 downto 0);
  signal memory_addr:         std_logic_vector(11 downto 0);

  -- internal signal for system bus
  signal system_addr:         std_logic_vector(31 downto 0);
  signal system_data_o:       std_logic_vector(31 downto 0);
  signal system_read:         std_logic;
  signal system_write:        std_logic;
  signal system_ack:          std_logic;

  -- internal signal for cpu
  signal cpu_data_i:          std_logic_vector(31 downto 0);
  signal cpu_addr_o:          std_logic_vector(31 downto 0);
  signal cpu_data_o:          std_logic_vector(31 downto 0);
  signal cpu_m_read:          std_logic;
  signal cpu_m_write:         std_logic;
  signal cpu_intack:          std_logic;
  signal cpu_ready_i:         std_logic;
  signal cpu_ack_o:           std_logic;

  -- internal signal for uart
  signal uart_ce:            std_logic;
  signal uart_addr:           std_logic_vector(1 downto 0);
  signal uart_data_i:         std_logic_vector(31 downto 0);
  signal uart_data_o:         std_logic_vector(31 downto 0);
  signal uart_rx_empty:       std_logic;
  signal uart_rx_irq:         std_logic;
  signal uart_tx_irq:         std_logic;
  signal uart_rxd:           std_logic;
  signal uart_txd:           std_logic;
  signal uart_cts:           std_logic;
  signal uart_rts:           std_logic;
```

CPU Component Binding

cpu1 is the eP32 CPU module instantiated in the eP32 chip. Its port map specifies how internal signals in cpu1 are connected to global signals in the chip system.

m_rst is inverted from the external master reset, arst. The external master reset, arst, is connected to a RESET pushbutton on the Brevia Board, and is normally pulled up to VCC. When RESET is pressed down, arst is pulled down to ground. Internal reset signals sent to the eP32 CPU and other memory and I/O devices use positive logic; therefore, arst must be inverted to m_rst, which is used to reset internal modules.

Here are local signals defined in the top level eP32 system. They are used to connect the eP32 CPU to other modules.

Local Signal	Function
m_rst	Master reset, inverted from external reset.
m_clk	Master clock, inverted from external clock to accommodate memory timing constraints.
system_addr	System address from CPU to all other modules.
system_read	Read enable from CPU to all other modules.
system_write	Write enable from CPU to all other modules.
system_ack	Acknowledge from CPU.
cpu_ready_i	CPU ready.
ready	System ready.
cpu_data_i	Data from another module to CPU. Individual byte is selected if the byte_word signal is set.
system_data_o	System data bus connected to memory and I/O modules. Memory and I/O devices are enabled by Bits 31-28 of system address.

UART Component Binding

The UART used in the eP32 is initialized to 115,200 baud, 1 start bit, 8 data bits, 2 stop bits, no parity, and no flow control. CTS and RTS, though defined in the UART module, are not used and not brought out to the eP32 package. Only RXD and TXD are brought out.

Local Signal	Function
uart_ce	UART enable
uart_addr	UART register address
uart_data_i	Data from CPU
uart_rxd	Receiver input
uart_txd	Transmitter output

```
  -- internal signal for gpio
  signal gpio_ce:             std_logic;
  signal gpio_addr:           std_logic_vector(1 downto 0);
  signal gpio_data_i:         std_logic_vector(31 downto 0);
  signal gpio_in:             std_logic_vector(15 downto 0);
  signal gpio_data_o:         std_logic_vector(31 downto 0);
  signal gpio_out:            std_logic_vector(15 downto 0);
  signal gpio_dir:            std_logic_vector(15 downto 0);

begin
  --
************************************************************************
*******
  --          Component Binding
  --
************************************************************************
*******
  -- ========================= CPU Block ============================
  cpu1: ep32
    port map (
      -- input port
      clk => aclk,
      clr => m_rst,
      interrupt => interrupt_i,
      data_i => cpu_data_i,
      intack => cpu_intack,
      read => cpu_m_read,
      write => cpu_m_write,
      addr => cpu_addr_o,
      data_o => cpu_data_o
    );

  --
************************************************************************
*******
  --          Internal Globle Signal Circuit
  --
************************************************************************
*******

  m_rst <= not arst;
  m_clk <= not aclk;
  system_addr <= cpu_addr_o;
  system_read <= cpu_m_read;
  system_write <= cpu_m_write;
  system_ack <= cpu_ack_o;
  cpu_ready_i <= '1';

  cpu_data_i <= system_data_o;

  system_data_o <=  cpu_data_o when (system_write='1') else
            memory_data_o when (system_addr(31 downto 28)="0000")
else
            uart_data_o when (system_addr(31 downto 28)="1000") else
            gpio_data_o when (system_addr(31 downto 28)="1110") else
            (others => 'Z');
```

RAM Component Binding

The RAM module handles only 32-bit words. Memory_addr, sent from CPU to memory modules, is at bits 11-0 to address 4k words of 32-bit word memory.

All other modules in the eP32 chip are clocked by the external master clock, aclk, except the RAM memory module, which is clocked by an inverted clock, m_clk. The reason is that the RAM_Q library module from Lattice IPexpress is a synchronous RAM memory, in which the rising edge of the clock latches the input address bus and input data bus. The eP32 expects asynchronous RAM/ROM memory modules, which must supply memory data to output to the data bus when the address bus is valid. All registers and stacks in the eP32 behave this way. Latching the address bus would waste one clock cycle for every memory access, making it impossible to execute all eP32 machine instructions in a single cycle.

A compromise between design specification and the available RAM_Q memory module is to clock RAM_Q modules with inverted clock m_clk, which forces latching the memory address bus a half-cycle earlier, on the rising edge of m_clk, which occurs on the falling edge of aclk. A disadvantage in clocking the memory address bus earlier is that the memory access speed must be twice the CPU speed. This is not a problem with FPGAs running at 50 MHz. Embedded RAM memory in FPGAs are generally much faster than 50 MHz. However, one should be careful in pushing CPU speed higher. You have to avoid contentions in accessing the memory bus.

Local Signal	Function
system_write	Write enable.
memory_addr	Word address sent to memory module.
memory_data_i	Data sent by CPU to memory module.
memory_data_o	Data output from memory module.

GPIO Component Binding

The GPIO module is defined as a 16-bit bidirectional I/O port. The gpio_idr signal can be used to change the I/O direction dynamically. However, in actual implementation, I/O devices used are switches, LED display, and LCD display. They do not require dynamic I/O redirection. In the eP32 system, gpio_in and gpio_out are merged into one ioport and brought to the eP32 package pins. io_port pins are defined as inout pins.

Local Signal	Function
gpio_ce	GPIO chip enable
gpio_addr	GPIO register address
gpio_data_i	Data send from CPU to GPIO module
gpio_in	Data received from GPIO input pins
ioport	16 bit bidirectional GPIO port

```
  -- ========================= UART Block
============================
  uart1 : uart
    port map (
      -- input
      clk_i => aclk,
      rst_i => m_rst,
      ce_i => uart_ce,
      read_i => system_read,
      write_i => system_write,
      addr_i => uart_addr,
      data_i => uart_data_i,
      -- output
      data_o => uart_data_o,
      rx_empty_o => uart_rx_empty,
      rx_irq_o => uart_rx_irq,
      tx_irq_o => uart_tx_irq,
      -- external interface
      rxd_i => uart_rxd,
      txd_o => uart_txd,
      cts_i => uart_cts,
      rts_o => uart_rts
    );
  uart_ce <= '1' when (system_addr(31 downto 28)="1000") else '0';
  uart_addr <= system_addr(1 downto 0);
  uart_data_i <= system_data_o;
  uart_rxd <= uart_i;
  uart_o <= uart_txd;

  -- ========================= RAM Block ============================
ram_memory_0 : ram_memory PORT MAP (
        Address  => memory_addr,
        Clock    => m_clk,
        ClockEn  => '1',
        Reset    => '0',
        Data     => memory_data_i,
        WE   => system_write,
        Q        => memory_data_o
    );

  memory_addr <= cpu_addr_o(11 downto 0);
  memory_data_i <= cpu_data_o ;
```

```
  -- ========================= GPIO Block
============================
  gpio1 : gpio
   port map (
      -- input port
      clr => m_rst,
      clk => aclk,
      write => system_write,
      read => system_read,
      ce => gpio_ce,
      addr => gpio_addr,
      data_in => gpio_data_i,
      gpio_in => gpio_in,
      -- output port
      data_out => gpio_data_o,
      gpio_out => gpio_out,
      gpio_dir => gpio_dir
    );
  gpio_ce <= '1' when (system_addr(31 downto 28)="1110") else
             '0';
  gpio_addr <= system_addr(1 downto 0);
  gpio_data_i <= system_data_o;
  gpio_in <= ioport;
  ioport(0)  <= gpio_out(0)  when gpio_dir(0)='1' else 'Z';
  ioport(1)  <= gpio_out(1)  when gpio_dir(1)='1' else 'Z';
  ioport(2)  <= gpio_out(2)  when gpio_dir(2)='1' else 'Z';
  ioport(3)  <= gpio_out(3)  when gpio_dir(3)='1' else 'Z';
  ioport(4)  <= gpio_out(4)  when gpio_dir(4)='1' else 'Z';
  ioport(5)  <= gpio_out(5)  when gpio_dir(5)='1' else 'Z';
  ioport(6)  <= gpio_out(6)  when gpio_dir(6)='1' else 'Z';
  ioport(7)  <= gpio_out(7)  when gpio_dir(7)='1' else 'Z';
  ioport(8)  <= gpio_out(8)  when gpio_dir(8)='1' else 'Z';
  ioport(9)  <= gpio_out(9)  when gpio_dir(9)='1' else 'Z';
  ioport(10) <= gpio_out(10) when gpio_dir(10)='1' else 'Z';
  ioport(11) <= gpio_out(11) when gpio_dir(11)='1' else 'Z';
  ioport(12) <= gpio_out(12) when gpio_dir(12)='1' else 'Z';
  ioport(13) <= gpio_out(13) when gpio_dir(13)='1' else 'Z';
  ioport(14) <= gpio_out(14) when gpio_dir(14)='1' else 'Z';
  ioport(15) <= gpio_out(15) when gpio_dir(15)='1' else 'Z';

  end behavioral;
```

5.2 The eP32 CPU Module

VHDL code of the eP32 CPU module is in the ep32.vhd file.

When I first learnt VHDL, the text books told me to build things in modules, to collect modules into libraries, and then call these modules out in the main design. So I did that in the original design of the P16. After a while, I found that the CPU was not that complicated, and all modules I needed could be combined together. The end result was that I had only one module and it is my entire CPU.

When RESET is set high, all registers and both stacks are cleared to 0. When RESET is cleared to 0, the CLOCK input drives the eP32. On the rising edge of CLOCK, the program word in memory address 0 is read into the I register. The first instruction in I is decoded; i.e., a set of control signals are sent to all components in the eP32. On the rising edge of the next CLOCK, new data are latched into appropriate registers and stacks depending on the instruction. The next instruction is decoded and thus executed, and so forth.

A memory interface is provided to connect to memory devices through a 32-bit address bus and a 32-bit data bus, with read enable and write enable control signals.

When reading a program word, the P register drives the external address bus and a program word is read into the I register. When reading or writing data words, the X register drives the external address bus, and data are read into the T register, or written from the T register, to the external data bus.

Two stacks are used in the eP32: a return stack to store return addresses from nested subroutine call instructions, and a data stack to store parameters passed among nested subroutines. The top two elements on the data stack are usually implemented as registers. They are the T register for “top”, and the S register for “second”. The top of the return stack is also implemented as the R register.

The T and S registers provide two inputs to the ALU, which carries out arithmetic and logic operations on data from T and S, and returns results to the T register.

The return stack, R, T, and S registers, and data stack can be viewed as a giant shift register array. Data can be shifted right or left in this giant array. The R, T and S registers are windows in this giant array visible to programmers in programming.

The eP32.vhd file contains the complete specification of this CPU in VHDL. You will be amazed at how simple a 32-bit CPU can be. I hope it will stimulate your mind, and encourage you to design you own dream microprocessor.

```
--
************************************************************
-- *            150nm Extreme Temperarture Radiation          *
-- *                Hardened SOC ASIC Project                 *
--
*============================================================*
-- * FPGA Project:       32-Bit CPU in Altera SOPC Builder      *
-- * File:              ep32.vhd                               *
-- * Author:            C.H.Ting                               *
-- * Description:        ep32 CPU Block                         *
-- *                                                          *
-- * Revision History:                                         *
-- * Date        By Who       Modification                     *
-- * 06/06/05    C.H. Ting    Convert EP24 to 32-bits.         *
-- * 06/10/05    Robyn King   Made compatible with Altera SOPC  *
-- *                         Builder.                         *
-- * 06/27/05    C.H. Ting    Removed Line Drawing Engine.     *
-- * 07/27/05    Robyn King   Cleaned up code.                 *
-- * 08/07/10    C. H. Ting   Return to eP32p                  *
-- * 11/18/10    Chen-Hanson Ting Port to LatticeXP2 Brevia Kit  *
--
************************************************************

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_misc.all;
use ieee.std_logic_unsigned.all;


entity ep32 is
    generic(width: integer := 32);
  port(
    -- input port
    clk:          in     std_logic;
    clr:          in     std_logic;
    interrupt:     in     std_logic_vector(4 downto 0);
    data_i:       in     std_logic_vector(31 downto 0);
    intack:       out    std_logic;
    read:         out    std_logic;
    write:        out    std_logic;
    addr:         out    std_logic_vector(31 downto 0);
    data_o:    out    std_logic_vector(31 downto 0)
  );
end entity ep32;
```

I/O Signals of the eP32 CPU

In VHDL terminology, the entity section specifies the interface signals from circuit component to the external world. The eP32, as a microprocessor chip, shows the pin-out of the chip in its entity section: master clock, control signals, data bus, address bus, and I/O ports. Here are detailed specifications of these busses and signals:

Signal	Function
clk	Master clock
clr	Master reset
interrupt	5-bit interrupt ports
data_i	32 bit data input bus
intack	Interrupt acknowledge
read	Memory read enable
write	Memory write enable
addr	32 bit address bus
data_o	32 bit data output bus

The eP32 CPU Module

An architecture section in VHDL is the body of the design, in which all internal signals and logic are contained. In an architecture section, signals and registers are defined first. Then there is a subsection where you can define concurrent logic, a subsection where you can define sequential logic, and a subsection defining the finite state machine that runs the show. For the purpose of documentation and clear referencing to signals, one can define constants to replace literal references.

```
architecture behavioral of ep32 is

    type stack is array(31 downto 0) of std_logic_vector(width downto
0);
    signal s_stack,r_stack: stack;
    signal slot: integer range 0 to 5;
    signal sp,sp1,rp,rp1: std_logic_vector(7 downto 0);
    signal t,s,sum: std_logic_vector(width downto 0);
    signal a,r: std_logic_vector(width downto 0);
    signal t_in,r_in,a_in: std_logic_vector(width downto 0);
    signal code: std_logic_vector(5 downto 0);
    signal t_sel: std_logic_vector(3 downto 0);
    signal p_sel: std_logic_vector(1 downto 0);
    signal a_sel: std_logic_vector(2 downto 0);
    signal r_sel: std_logic_vector(1 downto 0);
    signal addr_sel: std_logic;
    signal spush,spopp,rpush,rpopp,inten,intload,intset,
        tload,rload,aload,pload,iload,reset,z: std_logic;
    signal r_z,int_z: std_logic;
    signal i,p,p_in: std_logic_vector(width-1 downto 0);

  -- machine instructions selected by code

    constant bra : std_logic_vector(5 downto 0) :="000000";
    constant ret : std_logic_vector(5 downto 0) :="000001";
    constant bz  : std_logic_vector(5 downto 0) :="000010";
    constant bc  : std_logic_vector(5 downto 0) :="000011";

    constant call: std_logic_vector(5 downto 0) :="000100";
    constant nxt : std_logic_vector(5 downto 0) :="000101";
    constant ei  : std_logic_vector(5 downto 0) :="000110";

    constant ldp : std_logic_vector(5 downto 0) :="001001";
    constant ldi : std_logic_vector(5 downto 0) :="001010";
    constant ld  : std_logic_vector(5 downto 0) :="001011";

    constant stp : std_logic_vector(5 downto 0) :="001101";
    constant rr8 : std_logic_vector(5 downto 0) :="001110";
    constant st  : std_logic_vector(5 downto 0) :="001111";

    constant com : std_logic_vector(5 downto 0) :="010000";
    constant shl : std_logic_vector(5 downto 0) :="010001";
    constant shr : std_logic_vector(5 downto 0) :="010010";
    constant mul : std_logic_vector(5 downto 0) :="010011";

    constant xorr: std_logic_vector(5 downto 0) :="010100";
    constant andd: std_logic_vector(5 downto 0) :="010101";
    constant div : std_logic_vector(5 downto 0) :="010110";
    constant addd: std_logic_vector(5 downto 0) :="010111";

    constant popr: std_logic_vector(5 downto 0) :="011000";
    constant lda : std_logic_vector(5 downto 0) :="011001";
    constant dup : std_logic_vector(5 downto 0) :="011010";
    constant over: std_logic_vector(5 downto 0) :="011011";
```

Registers, Busses and Signals

Here are the registers, busses, and the internal control signals contained in the eP32 CPU. They are all defined as signals in VHDL. How they are actually implemented depends on how they are used in concurrent statements and in sequential statements.

Signal	Function
s_stack	Data stack.
r_stack	Return stack.
slot	Output of slot counter in finite state machine.
sp	Data stack pointer.
sp1	Alternate data stack pointer. It always has the value of sp+1.
rp	Return stack pointer.
rp1	Alternate return stack pointer. It always has the value of rp+1.
t	Accumulator or T register, top of data stack.
s	Top element of data stack. S is a pseudo register.
r	Top element of return stack. R is a real register.
a	Address register, X.
i	Instruction register, I.
p	Program counter, P.
sum	Output from an adder T+S.
t_in	Input to T register.
r_in	Input to R register.
a_in	Input to X register.
p_in:	Input to P register.
code	6-bit opcode decoded from I register.
spush	Control signals to push data stack.
spopp	Control signals to pop data stack.
rpush	Control signals to push return stack.
rpopp	Control signals to pop return stack.
tload	Enable signals to load T register.
aload	Enable signals to load X register.
pload	Enable signals to load P register.
iload	Enable signals to load I register.
z	One-bit signal, true if T=0, otherwise false.
r_z	One-bit signal, true if R=0, otherwise false.
int_z	One-bit signal, true if interrupt inputs are all 0, otherwise false.
inten	Enable interrupts.
intset	Set if interrupt is enabled
intload	Latch interrupt vector into P register.
a_sel	Select alternate argument to X register.
p_sel	Select alternate argument to P register.
r_sel	Select alternate argument to R register.
t_sel	Select alternate argument to T register.
addr_sel	Select alternate argument to address bus.

```
    constant pushr: std_logic_vector(5 downto 0) :="011100";
    constant sta : std_logic_vector(5 downto 0) :="011101";
    constant nop : std_logic_vector(5 downto 0) :="011110";
    constant drop: std_logic_vector(5 downto 0) :="011111";

-- mux to t register, selected by t_sel
    constant not_t: std_logic_vector :="0000";
    constant s_xor_t: std_logic_vector :="0001";
    constant s_and_t: std_logic_vector :="0010";
    constant s_or_t: std_logic_vector :="0011";
    constant sum_t: std_logic_vector :="0100";
    constant shr_sum: std_logic_vector :="0101";
    constant shr_t: std_logic_vector :="0110";
    constant shr_t_t: std_logic_vector :="0111";
    constant shl_sum_a_t: std_logic_vector :="1000";
    constant shl_t_a_t: std_logic_vector :="1001";
    constant shl_t: std_logic_vector :="1010";
    constant s_t: std_logic_vector :="1011";
    constant a_t: std_logic_vector :="1100";
    constant r_t: std_logic_vector :="1101";
    constant data_t: std_logic_vector :="1110";
    constant rr8_t: std_logic_vector :="1111";

-- mux to a register, selected by a_sel
    constant t_a: std_logic_vector :="001";
    constant a1_a: std_logic_vector :="010";
    constant shr_sum_a: std_logic_vector :="011";
    constant shr_t_a: std_logic_vector :="100";
    constant shl_sum_a: std_logic_vector :="101";

-- mux to r register, selected by r_sel
    constant rout_r: std_logic_vector :="00";
    constant t_r: std_logic_vector :="01";
    constant r1_r: std_logic_vector :="10";
    constant p_r: std_logic_vector :="11";

-- mux to p register, selected by p_sel
    constant pi_p: std_logic_vector :="00";
    constant p1_p: std_logic_vector :="01";
    constant r_p: std_logic_vector :="10";
    constant int_p: std_logic_vector :="11";

-- mux to memory bus, selected by addr_sel
    constant p_addr: std_logic :='0';
    constant a_addr: std_logic :='1';

begin

    data_o<= t(width-1 downto 0);
    intack <= inten;
    s <= s_stack(conv_integer(sp));

    sum <= (('0'&t(width-1 downto 0)) + ('0'&s(width-1 downto 0)));
```

Opcodes

Machine instructions, opcodes and their functions are as follows:

Instruction	Code	Function
bra	000000	Jump to address contained in current instruction.
ret	000001	Return from a subroutine to main program. Pop return address from return stack and store it in P.
bz	000010	If T=0, jump to address contained in current instruction; else continue.
bc	000011	If Carry is set, jump to address contained in current instruction; else continue.
call	000100	Push address in P on R stack, and jump to address contained in current instruction; else continue.
nxt	000101	If R is not 0, jump to address contained in current instruction, and decrement R by 1; else pop R stack and continue.
ei	000110	Enable interrupts.
ldp	001001	Push T on S stack, read memory word pointed to by X into T. Increment X by 1.
ldi	001010	Push T on S stack, read memory word pointed to by P into T. Increment P by 1.
ld	001011	Push T on S stack, read memory word pointed to by X into T.
stp	001101	Store T into memory pointed to by X. Increment X by 1. Pop S stack to T.
rr8	001110	Rotate T right by 8 bits.
st	001111	Store T into memory pointed to by X. Pop S stack to T.
com	010000	Complement T (1’s complement).
shl	010001	Shift T left by 1 bit.
shr	010010	Shift T right by 1 bit.
mul	010011	Multiplication step. If X(0)=1, add S to T, otherwise T is not changed. Shift T:X pair right by 1 bit.
xorr	010100	Pop S stack and XOR it to T.
andd	010101	Pop S stack and AND it to T.
div	010100	Division step. If T+S produces a carry, add S to T, otherwise T is not changed. Shift T:X pair left by 1 bit. Shift carry into X(0).
addd	010111	Pop S stack and add it to T.
popr	011000	Push T onto S stack. Pop R stack to T.
lda	011001	Push T onto S stack. Copy X to T.
dup	011010	Push T onto S stack.
over	011011	Push T onto S stack. Copy original contents of S to T.
pushr	011100	Push T onto R stack. Pop S stack to T.
sta	011101	Copy T to X. Pop S stack to T.
nop	011110	No operation.
drop	011111	Pop S stack to T.

```
    with t_sel select
    t_in <= (not t) when not_t,
        (t xor s) when s_xor_t,
        (t and s) when s_and_t,
        sum when sum_t,
        (t(width-1 downto 0) & '0') when shl_t,
        (t(width-1 downto 0) & a(width-1)) when shl_t_a_t,
        (sum(width-1 downto 0) & a(width-1)) when shl_sum_a_t,
        ('0'&sum(width downto 1)) when shr_sum,
        ('0'&t(width-1)&t(width-1 downto 1)) when shr_t,
        ("00"&t(width-1 downto 1)) when shr_t_t,
        s when s_t,
        a when a_t,
        r when r_t,
        t(width)&t(7 downto 0)&t(width-1 downto 8) when rr8_t,
        '0'&data_i(width-1 downto 0) when others;

    with slot select
    code <= i(29 downto 24) when 1,
        i(23 downto 18) when 2,
        i(17 downto 12) when 3,
        i(11 downto 6) when 4,
        i(5 downto 0) when 5,
        nop when others;
--  icode <= code;

    with a_sel select
    a_in <= a+1 when a1_a ,
        ('0'&t(0)&a(width-1 downto 1)) when shr_t_a ,
        ('0'&sum(0)&a(width-1 downto 1)) when shr_sum_a ,
        ('0'&a(width-2 downto 0)&sum(width)) when shl_sum_a ,
        t when others;

    with r_sel select
    r_in <= r-1 when r1_r ,
        '0'&p when p_r ,
        r_stack(conv_integer(rp)) when rout_r ,
        t when others;

    with p_sel select
    p_in <= (p(width-1 downto width-8) & i(width-9 downto 0)) when
pi_p ,
        r(width-1 downto 0) when r_p ,
        ("00000000000000000000000000000000"&interrupt(4 downto 0)) when
int_p ,
        p+1 when others;

    with addr_sel select
    addr <= a(width-1 downto 0) when a_addr ,
        p(width-1 downto 0) when others;
```

Concurrent Assignments

Most of the concurrent assignments (using "<=") simply route signals from one place to another. A few concurrent assignments actually do some useful things, like

Signal	Source
sum	Get sum of T+S.
z	z=1 if T=0; z=0 if T is not 0.
r_z	r_z=1 if R=0; r_z=0 if R is not 0.

The most interesting concurrent assignments are those of the multiplexers. Here are a few multiplexers explicitly defined, and their select signals:

Multiplexer	Select Signal
TMUX	t_sel
RMUX	r_sel
XMUX	a_sel
PMUX	p_sel
Address Bus	addr_sel
code	slot

The VHDL code on the left page shows constant values used to set selection signals to the various multiplexers.

Many other more complicated multiplexers are not defined explicitly, but are implicitly defined in case statements of individual machine instructions. Please examine these statements to see how particular signals are selected and routed.

data_o, which is the output data bus in the eP32 core, always sends out data in the T register. When we write data to memory and to peripheral devices, the address is provided in the X register, and data are provided in the T register.

"intack" is the interrupt acknowledge signal.

The S register is a pseudo-register. It is not defined as a register, but as the top of the data stack, s_stack, pointed to by the data stack pointer, sp. It is always used as the second argument, next to the T register, for arithmetic and logic machine instructions that expect two arguments.

"sum" is the adder in the eP32. It is shared by machine instructions ADD, MUL and DIV. It adds data from the T register and S register on the top of the data stack.

"t_in" is the output bus of a giant multiplexer, which provides input data to the T register. Machine instructions changing the T register must provide the proper select signal, t_sel, to this multiplexer to get the desired data routed to t_in. Then, on the rising edge of the next clock, data presented on t_in are latched into the T register.

```
    z <= not(t(width-1) or t(30) or t(29) or t(28)
        or t(27) or t(26) or t(25) or t(24)
        or t(23) or t(22) or t(21) or t(20)
        or t(19) or t(18) or t(17) or t(16)
        or t(15) or t(14) or t(13) or t(12)
        or t(11) or t(10) or t(9) or t(8)
        or t(7) or t(6) or t(5) or t(4)
        or t(3) or t(2) or t(1) or t(0));

    r_z <= not(r(width-1) or r(30) or r(29) or r(28)
        or r(27) or r(26) or r(25) or r(24)
        or r(23) or r(22) or r(21) or r(20)
        or r(19) or r(18) or r(17) or r(16)
        or r(15) or r(14) or r(13) or r(12)
        or r(11) or r(10) or r(9) or r(8)
        or r(7) or r(6) or r(5) or r(4)
        or r(3) or r(2) or r(1) or r(0));

    int_z <= interrupt(0) or interrupt(1) or interrupt(2)
        or interrupt(3) or interrupt(4) ;

  -- sequential assignments, with slot and code
    decode: process(code,a,z,r_z,int_z,t,slot,sum,inten) begin
        t_sel<="0000";
        a_sel<="000";
        p_sel<="00";
        r_sel<="00";
        addr_sel<='0';
        spush<='0';
        spopp<='0';
        rpush<='0';
        rpopp<='0';
        tload<='0';
        aload<='0';
        pload<='0';
        rload<='0';
        write<='0';
        read<='0';
        iload<='0';
        reset<='0';
        intload<='0';
        intset<='0';

    if slot=0 then
        if (int_z='1' and inten='1') then
            pload<='1';
            p_sel<=int_p;--process interrupts
            rpush<='1';
            r_sel<=p_r;
            rload<='1';
            reset<='1';
        else   iload<='1';
            p_sel<=p1_p;--fetch next word
            pload<='1';
           read<='1';
        end if;
    else
```

“code” is the output bus of the instruction multiplexer, which selects one of 5 machine instructions stored in the I register. “slot” selects the machine instruction to be executed in the current clock cycle. “code” will be used in the instruction decoder’s decode process, to produce relevant control signals to execute the selected machine instruction.

“a_in” is the input bus of the XMUX multiplexer, which normally gets data from the T register. However, when executing memory read/write instructions, it can optionally increment by selecting data from the X register through an increment circuit. Used in MUL and DIV instructions, it takes data from the X register shifted to the right or left, respectively. Shifting operations are coordinated with the T register so that the T:X register pair acts like a 65-bit shift register.

“r_in” is the input bus of the R register, which selects data from the P register for the CALL instruction, the T register for the PUSHR instruction, the top of the return stack r_stack for the POPR instruction, and from R-1 for the NEXT instruction. It manages the return stack in the eP32.

“p_in” is the input bus of the P register, which selects data from P+1 in slot0 to fetch the next program word, the R register for the RET instruction. In slot0, if interrupt pins are not all zero and when interrupts are enabled, p_in selects 5 bits from the interrupt input pins, zero extended to 32 bits, to jump to an interrupt service routine.

“addr” is the output bus of the address multiplexer, which provides addresses to output bus addr_o of the eP32 module. It outputs address in the P register when reading program words, or addresses in the X register when reading and writing data to/from memory or peripheral devices.

“z” returns a 1 if bits T(0) to T(31) are all zero. If any of these bits is not a zero, z returns a zero. It is used by the BZ instruction to branch to a new program location when T is zero.

“r_z” returns a 1 if bits R(0) to R(31) are all zero. If any of these bits are not a zero, r_z returns a zero. It is used by the NEXT instruction to loop to a new program location when R is zero. It allows looping in a single clock cycle.

“int_z” returns a 1 if bits interrupt(0) to interrupt(4) are all zero. If any of these bits are not a zero and interrupts are enabled, a jump is made to an interrupt service routine.

```
    case code is
        when bra =>
            pload<='1';
            p_sel<=pi_p;
            reset<='1';
        when ret => pload<='1';
            p_sel<=r_p;
            rpopp<='1';
            r_sel<=rout_r;
            rload<='1';
            reset<='1';
            intset<='0';
            intload<='1';
        when bz =>
            if z='1' then
                pload<='1';
                p_sel<=pi_p;
            end if;
            tload<='1';
            t_sel<=s_t;
            spopp<='1';
            reset<='1';
        when bc =>
            if t(width)='1' then
                pload<='1';
                p_sel<=pi_p;
            end if;
            tload<='1';
            t_sel<=s_t;
            spopp<='1';
            reset<='1';
        when call =>
            pload<='1';
            p_sel<=pi_p;--process call
            rpush<='1';
            r_sel<=p_r;
            rload<='1';
            reset<='1';
        when nxt =>
            if r_z='0' then
                p_sel<=pi_p;
                pload<='1';
                r_sel<=r1_r;
            else
                r_sel<=rout_r;
                rpopp<='1';
            end if;
            rload<='1';
            reset<='1';
        when ei =>
            intset<='1';
            intload<='1';
```

Sequential Assignments

This big sequential assignment is the instruction decoder of the eP32 CPU. In the "decode" process, control signals are initialized and then set according to the needs of each different machine instruction. These control signals flow out to concurrent assignments to select proper signals to be latched into registers and stacks, on the rising edge of the next clock pulse.

When slot=0, that is, the slot machine is executing a slot0 function, the external 5 bit interrupt signals are examined. If all interrupt signals are low, the address of the next program word in the P register is sent out to the address bus. "iload" is set so that a program word from the external data bus will be latched into the I register. "pload" is also set so that the P register will be incremented.

If any bit of the interrupt signals is high, then a subroutine call is forced to an address from location 1 to 31, as specified by the 5-bit interrupt input signals.

If "slot" is not zero, then a machine code in slot1 to slot5 of the I register is selected and executed. Executing a machine instruction is simply setting some control signals to route proper data through concurrent logic and connecting multiplexers to targeted registers and stacks. On the rising edge of the next master clock, all data are latched and then the next machine instruction is decoded and executed.

First, default values of signals are assigned. In all instructions, only a few of these signals are changed to achieve specific functions, and we only have to specify those changed signals for those instructions.

Here are the signals changed when the instruction sequencer is in Slot0. This includes external interrupt pins. If one or more interrupts are set, the CPU calls an interrupt service routine from memory location 1 to 31. If no interrupt is set, this causes the program word pointed to by the P register to be fetched, and the instruction sequencer is incremented to Slot1, in preparation to execute the first instruction in the program word.

If there is an interrupt request, call an interrupt vector.

Signal	Function
pload<='1'	Load P register
p_sel<=int_p	Select interrupt vector for P register
rpush<='1'	Push P to R and return stack
r_sel<=p_r	Select P for RMUX
rload<='1'	Load R register
reset<='1'	Force next cycle to slot0

If there is no interrupt request, fetch and execute the next program word.

Signal	Function
iload<='1'	Load I register
p_sel<=p1_p	Select P+1 to P register
pload<='1'	Load P register
read<='1'	Read program memory to P register

```
        when ldp => addr_sel<=a_addr;
            a_sel<=a1_a;
            aload<='1';
            tload<='1';
            t_sel<=data_t;
            spush<='1';
           read<='1';
        when ldi => pload<='1';
            p_sel<=p1_p;
            tload<='1';
            t_sel<=data_t;
            spush<='1';
           read<='1';
        when ld => addr_sel<=a_addr;
            tload<='1';
            t_sel<=data_t;
            spush<='1';
           read<='1';
        when stp => addr_sel<=a_addr;
            aload<='1';
            a_sel<=a1_a;
            tload<='1';
            t_sel<=s_t;
            spopp<='1';
            write<='1';
        when st => addr_sel<=a_addr;
            tload<='1';
            t_sel<=s_t;
            spopp<='1';
            write<='1';
        when rr8 =>
            tload<='1';
            t_sel<=rr8_t;
        when com =>
            tload<='1';
            t_sel<=not_t;
        when shl =>
            tload<='1';
            t_sel<=shl_t;
        when shr =>
            tload<='1';
            t_sel<=shr_t;
        when mul =>
            aload<='1';
            tload<='1';
            if a(0)='1' then
                t_sel<=shr_sum;
                a_sel<=shr_sum_a;
            else    t_sel<=shr_t_t;
                a_sel<=shr_t_a;
            end if;
        when xorr =>
            tload<='1';
            t_sel<=s_xor_t;
            spopp<='1';
```

Decoder

The big case statement using “code” as selector determines which machine instruction to execute, which control signals are set or cleared, which signals must go through their respective multiplexers, and which signals are to be latched into registers and stacks.

If the instruction sequencer is not in Slot0, it executes instruction “code” selected from one of 5 slots in the I register. This is a giant case statement listing all changed signals associated with each and every instruction. These instructions change appropriate signals to route proper signals through busses and multiplexers, to be latched into stacks and registers on the rising edge of the next clock.

Transfer Instructions

Following are transfer instructions, which load a target program address into the P register, and thus jump to different memory locations. The target address is formed by appending the contents of the address field of the long instruction to the 8-bit page address in the P register. Therefore transfer instructions can branch to any location within the current 16M word page. Only the RET instruction can branch to the entire 32-bit memory space, because it obtains its target address from the R register.

To execute the BRA instruction, set the following signals:

pload<='1'	Load P register
p_sel<=pi_p	Select address field for P register
reset<='1'	Force next cycle to slot0

To execute the RET instruction, set the following signals:

pload<='1'	Load P register
p_sel<=r_p	Select R register to load P register
rpopp<='1'	Pop return stack
r_sel<=rout_r	Select r_stack to load R register
rload<='1'	Load R register
reset<='1'	Force next cycle to slot0
intset<='0'	Clear interrupt enable flag
intload<='1'	Load inten register

To execute the BZ instruction, set the following signals if T=0:

pload<='1'	Load P register
p_sel<=pi_p	Select address field for P register

Always set the following signals:

tload<='1'	Load T register
t_sel<=s_t	Select top of s_stack to load T register
spopp<='1'	Pop s_stack
reset<='1'	Force next cycle to slot0

```
        when andd =>
            tload<='1';
            t_sel<=s_and_t;
            spopp<='1';
        when div =>
            aload<='1';
            tload<='1';
            a_sel<=shl_sum_a;
            if sum(width)='1' then
                t_sel<=shl_sum_a_t;
            else    t_sel<=shl_t_a_t;
            end if;
        when addd =>
            tload<='1';
            t_sel<=sum_t;
            spopp<='1';
        when popr =>
            tload<='1';
            t_sel<=r_t;
            spush<='1';
            r_sel<=rout_r;
            rload<='1';
            rpopp<='1';
        when lda =>
            tload<='1';
            t_sel<=a_t;
            spush<='1';
        when dup =>
            spush<='1';
        when over =>
            spush<='1';
            tload<='1';
            t_sel<=s_t;
        when pushr =>
            tload<='1';
            t_sel<=s_t;
            rpush<='1';
            r_sel<=t_r;
            rload<='1';
            spopp<='1';
        when sta =>
            tload<='1';
            t_sel<=s_t;
            a_sel<=t_a;
            aload<='1';
            spopp<='1';
        when nop => reset<='1';
        when drop =>
            tload<='1';
            t_sel<=s_t;
            spopp<='1';
        when others => null;
    end case;
    end if;
    end process decode;
```

To execute the BC instruction, set the following signals if carry T(32)=1:

pload<='1'	Load P register
p_sel<=pi_p	Select address field for P register

Always set the following signals:

tload<='1'	Load T register
t_sel<=s_t	Select top of s_stack to load T register
spopp<='1'	Pop s_stack
reset<='1'	Force next cycle to slot0

To execute the CALL instruction, set the following signals:

pload<='1'	Load P register
p_sel<=pi_p	Select address field for P register
rpush<='1'	Push R and r_stack
r_sel<=p_r	Select P to load R register
rload<='1'	Load R register
reset<='1'	Force next cycle to slot0

The NXT instruction is probably the most complicated transfer instruction. It is a single cycle loop instruction. It uses the R register as a loop counter, counting down towards 0. When R is not zero, it is decremented, and program register P is loaded with an address in the address field of this long transfer instruction. The loop is then repeated. When R is decremented to 0, the R register and r_stack are popped, and execution continues with the next program word. The loop is thus terminated.

To execute the NXT instruction, set the following signals if R is not 0:

p_sel<=pi_p	Select address field for P register
pload<='1'	Load P register
r_sel<=r1_r	Load R-1 into R register

Set the following signals if R is 0:

r_sel<=rout_r	Select top of r_stack to load R register
rpopp<='1'	Pop r_stack

Always set the following signals:

rload<='1'	Load R register
reset<='1'	Force next cycle to slot0

Enable Interrupts

To execute the EI instruction, set the following signals:

intset<='1'	Set interrupt acknowledge flag
intload<='1'	Load inten (interrupt enable) register

Memory Instructions

Following are the memory instructions, which read data from memory to the T register or write data from the T register to memory. The address of memory is always in the X register. When reading, the T register is pushed onto the data stack. When writing, the data stack is popped to the T register.

To execute the LDP instruction, set the following signals:

addr_sel<=a_addr	Select X to load memory address bus
a_sel<=a1_a	Increment X register
aload<='1'	Load X register
tload<='1'	Load T register
t_sel<=data_t	Select data bus to load T register
spush<='1'	Push s_stack
read<='1'	Enable memory read

To execute the LDI instruction, set the following signals:

pload<='1'	Load P register
p_sel<=p1_p	Select P+1 to load P register
tload<='1'	Load T register
t_sel<=data_t	Select data bus to load T register
spush<='1'	Push s_stack
read<='1'	Enable memory read

To execute the LD instruction, set the following signals:

addr_sel<=a_addr	Select X to load memory address bus
tload<='1'	Load T register
t_sel<=data_t	Select data bus to load T register
spush<='1'	Push s_stack
read<='1'	Enable memory read

To execute the STP instruction, set the following signals:

addr_sel<=a_addr	Select X to load memory address bus
aload<='1'	Load X register
a_sel<=a1_a	Increment X register
tload<='1'	Load T register
t_sel<=s_t	Select R to load T register
spopp<='1'	Pop s_stack
write<='1'	Enable memory write

To execute the ST instruction, set the following signals:

addr_sel<=a_addr	Select X to load memory address bus
tload<='1'	Load T register
t_sel<=s_t	Select R to load T register
spopp<='1'	Pop s_stack
write<='1'	Enable memory write

ALU Instructions

To execute the RR8 instruction, set the following signals:

tload<='1'	Load T register
t_sel<=rr8_t	Select T rotate right 8 bit to load T register

To execute the ST instruction, set the following signals:

tload<='1'	Load T register
t_sel<=not_t	Select not(T) to load T register

To execute the SHL instruction, set the following signals:

tload<='1'	Load T register
t_sel<=shl_t	Shift T left 1 bit

To execute the SHR instruction, set the following signals:

tload<='1'	Load T register
t_sel<=shr_t	Shift T right 1 bit

To execute the XOR instruction, set the following signals:

tload<='1'	Load T register
t_sel<=s_xor_t	Select (S xor T) to load T register
spopp<='1'	Pop s_stack

To execute the AND instruction, set the following signals:

tload<='1'	Load T register
t_sel<=s_and_t	Select (S and T) to load T register
spopp<='1'	Pop s_stack

To execute the ADD instruction, set the following signals:

tload<='1'	Load T register
t_sel<=sum_t	Select (S + T) to load T register
spopp<='1'	Pop s_stack

MUL Step

The MUL step and DIV step instructions are the most complicated instructions. They use T and X as a register pair. The T-X register pair is shifted right or left, and the T register may either receive results from the adder or remain unchanged. Repeating these instructions is the simplest and the most efficient way to implement an unsigned multiplier and an unsigned divider.

In the MUL instruction, the T and X registers are considered a 65-bit right-shift register. Initially, a partial sum is loaded in the T register, a multiplier in the X register, and a multiplicand in the S register. If the least significant bit in X is 1, S is added to T, and the resulting T-X pair is shifted right by 1 bit. If the least significant bit in X is 0, T is not changed, and the T-X pair is shifted right by 1 bit. After repeating the MUL instruction 32 times, the T-X register pair will contain a double product of X*S +T.

To execute the MUL instruction when X(0)=1 :

aload<='1'	Load X register
tload<='1'	Load T register
t_sel<=shr_sum	Select right shifted (S+T):X
a_sel<=shr_sum_a	Select right shifted (S+T):X

To execute the MUL instruction when X(0)=0 :

aload<='1'	Load X register
tload<='1'	Load T register
t_sel<=shr_t_t	Select right shifted T:X
a_sel<=shr_t_a	Select right shifted T:X

DIV Step

In the DIV instruction, the T and X registers are again considered a 65-bit left-shift register. A double integer dividend is contained in the T-X register pair, and a negated divisor is in the S register. In the ALU, the sum of S and T is always computed by an adder. If the carry bit in adder sum(32) is 1, S is added to T, and the resulting T-X pair is shifted left by 1 bit. If the carry bit in adder is 0, T is not changed, and the T-X register pair is shifted left by 1 bit. In either case, the carry bit is shifted into the least significant bit in the X register. After repeating the DIV instruction 33 times, the X register contains the quotient, and the T register contains 2x of the remainder of division.

To execute the DIV instruction when the carry bit sum(32)=1 :

aload<='1'	Load X register
tload<='1'	Load T register
a_sel<=shl_sum_a	Select left shifted T:X
t_sel<=shl_sum_a_t	Select left shifted (S+T):X

To execute the DIV instruction when the carry bit sum(32)=0 :

aload<='1'	Load X register
tload<='1'	Load T register
a_sel<=shl_sum_a	Select left shifted T:X
t_sel<=shl_t_a_t	Select left shifted T:X

Register and Stack Instructions

To execute the POPR instruction, set the following signals:

tload<='1'	Load T register
t_sel<=r_t	Select R to load T register
spush<='1'	Push s_stack
r_sel<=rout_r	Select r_stack to load R register
rload<='1'	Load R register
rpopp<='1'	Pop r_stack

To execute the XT instruction, set the following signals:

tload<='1'	Load T register
t_sel<=a_t	Select X to load T register
spush<='1'	Push s_stack

To execute the DUP instruction, set the following signals:

spush<='1'	Push s_stack

To execute the OVER instruction, set the following signals:

spush<='1'	Push s_stack
tload<='1'	Load T register
t_sel<=s_t	Select S to load T register

To execute the PUSHR instruction, set the following signals:

tload<='1'	Load T register
t_sel<=s_t	Select S to load T register
rpush<='1'	Push r_stack
r_sel<=t_r	Select T to load R register
rload<='1'	Load R register
spopp<='1'	Pop s_stack

To execute the TX instruction, set the following signals:

tload<='1'	Load T register
t_sel<=s_t	Select S to load T register
a_sel<=t_a	Select T to load X register
aload<='1'	Load X register
spopp<='1'	Pop s_stack

To execute the NOP instruction, set the following signals:

reset<='1'	Force next cycle to slot0

To execute the DROP instruction, set the following signals:

tload<='1'	Load T register
t_sel<=s_t	Select S to load T register
spopp<='1'	Pop s_stack

```
-- finite state machine, processor control unit
    sync: process(clk,clr) begin
        if clr='1' then -- master reset
            inten <='0';  slot <= 0;
            sp  <= "00000000";  sp1 <= "00000001";
            rp  <= "00000000";  rp1 <= "00000001";
            t <= (others => '0');
            r <= (others => '0');
            a <= (others => '0');
            p <= (others => '0');
            i <= (others => '0');
            for ii in s_stack'range loop
                s_stack(ii) <= (others => '0');
                r_stack(ii) <= (others => '0');
            end loop;
        elsif (clk'event and clk='1') then
            if reset='1' or slot=5 then
                slot <= 0;
            else    slot <= slot+1;
            end if;
            if intload='1' then
                inten <= intset;
            end if;
            if iload='1' then
                i <= data_i(width-1 downto 0);
            end if;
            if pload='1' then
                p <= p_in;
            end if;
            if tload='1' then
                t <= t_in;
            end if;
            if rload='1' then
                r <= r_in;
            end if;
            if aload='1' then
                a <= a_in;
            end if;
            if spush='1' then
                s_stack(conv_integer(sp1)) <= t;
                sp <= sp+1;
                sp1 <= sp1+1;
            elsif spopp='1' then
                sp <= sp-1;
                sp1 <= sp1-1;
            end if;
            if rpush='1' then
                r_stack(conv_integer(rp1)) <= r;
                rp <= rp+1;
                rp1 <= rp1+1;
            elsif rpopp='1' then
                rp <= rp-1;
                rp1 <= rp1-1;
            end if;
        end if;
    end process sync;
end behavioral;
```

Finite State Machine

Finite state machine “sync” is a process paced by master clock “clk”. This is what I called a Slot Machine. The master clock drives a 6-state counter, “slot”, and increments it from 0 to 5 and then repeats the sequence. Each clock cycle can thus be named slot0 to slot5, according to the contents of “slot”.

Machine instructions are decoded in the “decode” process, where control and select signals are set and data are routed through concurrent logic and multiplexers. On the rising edge of master clock “clk”, selected registers and stacks latch outputs from respective multiplexers. A machine instruction is thus executed. The “slot” counter is incremented, and the next instruction from the next slot in the I register is decoded and executed.

When “slot” is 5, or when a transfer instruction (CALL, RET, BRA, BZ, or BNC) is executed, the counter “slot” is cleared to 0. In the next clock cycle, slot0, the eP32 will process an interrupt if any interrupt is pending, or fetch the next program word from memory and start executing machine instructions contained in this program word.

When “clr” is set, the eP32 is in a reset state. In the reset state, all registers and both stacks are cleared to 0, except sp1 and rp1, which are initialized to 1. When “clr” is cleared to 0, the eP32 starts running. Since the P register is cleared to 0 on reset, and “slot” is 0, the program word in memory location 0 is fetched from memory on the rising edge of master clock “clk”. On the rising edge of the next clock, the machine instruction in slot1 of this program word is decoded and executed. What happens next depends on this instruction.

All elements in s_stack and r_stack are cleared using a for-loop in the sync process.

When “clr” is cleared to 0, the master clock starts driving the Slot Machine and starts the CPU running. (clk'event and clk='1') specifies that all actions occur on the rising edge of master clock “clk”.

On the rising edge of “clk”, the counter “slot” is incremented. When “slot” is incremented to 5, or when reset=1, as a transfer instruction (CALL, RET, BRA, BZ, or BNC) is executed, “slot” is cleared to 0. In the next clock cycle, slot0, the eP32 will process an interrupt if any interrupt is pending, or fetch the next program word from memory and start executing machine instructions contained in this program word.

If intload=1, the inten register is aligned to intset, which enables or disables interrupts.
If iload=1, the next program word is latched into I register.
If pload=1, the P register is loaded from PMUX.
If tload=1, the T register is loaded from TMUX.
If rload=1, the R register is loaded from RMUX.
When aload=1, the X register is loaded from XMUX.

The data stack and return stack are implemented as 32 33-bit register arrays. Stacks

have to be pushed or popped in a single clock cycle, with all other actions in the CPU. When pushing, the stack pointer must be pre-incremented, and when popping, the stack pointer must be post-decremented. In conventional designs, it would take another cycle to pre-increment a stack pointer. To make sure that all stack actions are always accomplished in a single cycle, we add two auxiliary stack pointers, sp1 and rp1, which are always one count above the principal stack pointers, sp and rp. When pushing, sp1 or rp1 is used to write a new stack element above the top of stack. When popping, sp or rp is used to read the top element on the stack. Whenever sp or rp is changed, sp1 or rp1 are changed accordingly, too.

When pushing the data stack, spush=1. The T register is copied to the top of s_stack, pointed to by sp1. This is what is called pre-incrementing, as sp1 is pointing to a location above the top of the data stack, pointed to by sp. Then, both sp and sp1 are incremented, so that now sp is pointing to the new location on top of s_stack.

When popping the data stack, spopp=1. Nothing in particular needs to be done, as the top of s_stack pointed to by sp is read out. On the rising edge of the next clock, both sp and sp1 are decremented. This is post-decrementing.

When pushing the return stack, rpush=1. The R register is copied to the top of r_stack, pointed to by rp1. rp1 is pointing to a location above the top of the return stack, pointed to by rp. Then, both rp and rp1 are incremented, so that now rp is pointing to the new location on the top of r_stack.

When popping the return stack, rpopp=1. The top of the r_stack pointed to by rp is read out. On the rising edge of the next clock, both rp and rp1 are decremented.

5.3 RAM Memory Module

The VHDL code of the RAM module is in the ram_memory.vhd file.

The design of the memory module is different for FPGAs from different manufacturers. It is the only module in the eP32 that cannot be ported across FPGA chips. However, FPGA manufacturers generally supply memory blocks in VHDL and Verilog modules. The user can pick the memory block from a library, and configure it to suit his design requirements. Some FPGA systems allow the user to initialize a memory block so that the resulting microprocessor system can boot up immediately on power up.

For the eP32 system, the memory block has to be configured as follows:
Memory word width 32 bits
Memory depth 4096 or more words
Single phase clock
No input latch
No output latch

Some FPGAs contain ROM and RAM memory blocks. ROM memory must be initialized to contain program code. The LatticeXP2 has only RAM memory blocks, but RAM memory is initialized from flash memory. This configuration is very convenient for microprocessor designs, because the microprocessor can be initialized

immediately from flash memory on power up, and programs are executed in RAM. No extra ROM memory is necessary to store program code, and a single FPGA chip becomes a complete microprocessor system.

The eForth system software to be executed on the eP32 chip must be compiled and copied into an mem.mif file. mem.mif must be copied into the eP32 project folder so that the ispLEVER system can use it to initialize RAM memory. When the eP32 chip design is downloaded into a LatticeXP2 chip, eForth goes along.

The eP32 uses memory of the simplest type, asynchronous RAM memory. No clock signal is needed for reading. When the address bus is stable, the addressed memory cell puts its contents on the output data bus. When memory is in write mode, write-enable is pulled high. Then when the write-clock pulse is high, input data on the data bus is written into the memory cell addressed by the address bus. This is how most static RAM memory chips were designed and implemented. Most FPGA manufacturers, however, choose to implement their RAM modules as synchronous RAM, which uses a clock pulse to first latch its address and data bus, and then put the addressed memory cell on the output data bus.

One must be very careful in clocking memory blocks. Synchronous memory is incompatible with the eP32 design, because the memory contents are not available before the rising clock edge, after the memory address is changed. In the eP32, memory contents must be stable before the rising clock edge. This clocking problem is solved by using synchronous memory blocks and clocking them with the trailing edge of the master clock. A disadvantage is that the CPU can only run at 1/2 of maximum memory access speed. It is not a problem with most FPGAs running at 50 MHz. It may become a problem when you have to push the speed higher.
A few lines of data in mem.mif in Addressed-Hex format are as follows:

```
#Format=AddrHex
#Depth=4096
#Width=32
#AddrRadix=3
#DataRadix=3
#Data
0:68D
24:80
25:A
26:7C6
27:7C8
28:7C6
29:4A0
2A:4D2
2D:7C6
101:564F4405
102:5241
103:1805E79E
104:101
105:3C3002
106:1179E79E
107:3000109
108:1A69405E
109:A05E79E
10A:FFFFFFFF
10B:105
```

```
10C:2B4D5503
10D:1769E79E
10E:3000110
10F:1A69405E
110:A05E79E
111:1
112:10C
113:55443F04
114:50
115:1A79E79E
116:2000118
117:1A05E79E
118:179E79E
119:113
11A:454E4407
11B:45544147
11C:10710297
11D:1
11E:1A79E79E
11F:3000121
120:1805E79E
121:1829705E
122:1
123:11A
124:53424103
125:1A45E79E
```

```
126:3000128
127:179E79E
128:1029705E
```

Only the first page of ram_memory.vhd is shown on the left page. It is generated automatically by the RAM_Q memory module in the IPexpress library of the ispLEVER system. Terms used in this file are incomrehensible except to experts at Lattice, and I will not try to comment on it. We just need to know its interface to the eP32, and leave the details to Lattice and the ispLEVER system.

RAM memory is mapped in the the address space between 0 and 0xFFF.

Port signals defined for the RAM memory module are:

Port Signal	Function
address	Address from CPU
clock	Memory clock, inverted from master clock
clockEn	Clock enable, always enabled
data	Data input from CPU
reset	Clear addfress and data registers, always disabled
we	Write enable from CPU
q	Data output to CPU

VHDL code for this memory module is generated automatically by IPexpress in the ispLEVER system. It is not printed here.

RAM memory must be initialized properly with a program in it, so that when the eP32 chip is synthesized and downloaded into the FPGA, the program starts executing after Reset is released and the clock is applied to the chip. RAM memory is initialized with the contents of the mem.mif file. This file is produced by the eForth metacompiler, which builds a memory image of the eForth system, and copies this image into the mem.mif file. The mem.mif file must be copied into the folder where all other VHDL files reside. When IPexpress in the ispLEVER System generates mem_memory.vhd, it reads mem.mif and includes code instantiating program words into the RAM module.

```
-- VHDL netlist generated by SCUBA ispLever_v81_SP1_Build (36)
-- Module  Version: 7.1
--D:\ispTool\ispfpga\bin\nt\scuba.exe -w -lang vhdl -synth synplify
-bus_exp 7 -bb -arch mg5a00 -type bram -wp 10 -rp 1000 -addr_width
12 -data_width 32 -num_rows 4096 -writemode NORMAL -resetmode SYNC
-memfile
d:/isptool/demo_latticexp2_brevia_soc_vhdl/demo_latticexp2_brevia
_soc/project/ep32q_xp2_4/mem.mif -memformat orca -cascade -1 -e

-- Sat Dec 11 08:41:47 2010

library IEEE;
use IEEE.std_logic_1164.all;
-- synopsys translate_off
library xp2;
use xp2.components.all;
-- synopsys translate_on

entity ram_memory is
    port (
        Clock: in  std_logic;
        ClockEn: in  std_logic;
        Reset: in  std_logic;
        WE: in  std_logic;
        Address: in  std_logic_vector(11 downto 0);
        Data: in  std_logic_vector(31 downto 0);
        Q: out  std_logic_vector(31 downto 0));
end ram_memory;

architecture Structure of ram_memory is

    -- internal signal declarations
    signal scuba_vhi: std_logic;
    signal scuba_vlo: std_logic;

    -- local component declarations
    component VHI
        port (Z: out  std_logic);
    end component;
    component VLO
        port (Z: out  std_logic);
    end component;
    component DP16KB
    -- synopsys translate_off
        generic (INITVAL_3F : in String; INITVAL_3E : in String;
                INITVAL_3D : in String; INITVAL_3C : in String;
                INITVAL_3B : in String; INITVAL_3A : in String;
                INITVAL_39 : in String; INITVAL_38 : in String;
                INITVAL_37 : in String; INITVAL_36 : in String;
                INITVAL_35 : in String; INITVAL_34 : in String;
                INITVAL_33 : in String; INITVAL_32 : in String;
                INITVAL_31 : in String; INITVAL_30 : in String;
                INITVAL_2F : in String; INITVAL_2E : in String;
                INITVAL_2D : in String; INITVAL_2C : in String;
                INITVAL_2B : in String; INITVAL_2A : in String;
                INITVAL_29 : in String; INITVAL_28 : in String;
                INITVAL_27 : in String; INITVAL_26 : in String;
                INITVAL_25 : in String; INITVAL_24 : in String;
```

5.4 UART Module

The VHDL code of the UART module is in the uart.vhd file.

A UART port is the simplest and the most efficient I/O device allowing a FORTH system to interact with users. With a UART port, we can bring up an eP32 system on power-up and a user can immediately begin software development.

This UART system is set to 115,200 baud, 1 start bit, 8 data bits, 1 stop bit, no parity, and no flow control.

4 Registers are defined in the UART module, and their addresses and functions are as follows:

Address	Register	Function
Ox80000000	Baud Rate Register	32-bit baud rate counter
Ox80000001	Transmit Register	Bits7-0, transmit data; bit8, transmitter status
Ox80000002	Receive Status Register	Bit0, flow control, bit8 receiver status
Ox80000003	Receive Buffer Register	Bits7-0, Receive data

Signals in UART modules are defined in an architecture as follows:

Port Signal	Function
clk_i	Master clock input
rst_i	Master reset input
ce_i	UART chip select input
read_i	Read enable input
write_i	Write enable input
addr_i	Register address input
data_i	Data input from CPU
data_o	Data output to CPU
rx_empty_o	Receiver buffer empty
rx_irq_o	Receiver interrupt request
tx_irq_o	Transmitter interrupt request
rxd_i	Receiver data input
txd_o	Transmitter data output
cts_i	Clear-to-Send input
rts_o	Ready-to-Send output

The UART is initialized to run at 115,200 baud. Using a 50 MHz crystal for the master clock, the baud rate register is set to 431. When I switched to a 16 MHz clock, the board seemed to work fine at 38,400 baud. UART devices are very forgiving in clock variations. The baud rate register is a read-write register, and baud rate can be dynamically changed by writing a new baud rate count into the baud rate register.

```
************************************************************
-- *           UART Serial Interface                .      *
-- *==========================================================*
-- * Project:          FG in PROASIC                         *
-- * File:             uart.vhd                              *
-- * Author:           Chien-Chia Wu                         *
-- * 02/13/03  Chien-Chia Wu   Reference uart statements t   *
-- * 02/14/03  Chien-Chia Wu   (1)Copy from bpchip,          *
-- *                           (2)Modify to 32-bits   .      *
-- *                           (3)Swap the cts and rts .     *
-- ************************************************************
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_misc.all;
use ieee.std_logic_unsigned.all;
entity uart is
    port(
        -- input
        clk_i:      in      std_logic;
        rst_i:      in      std_logic;
        ce_i:       in      std_logic;
        read_i:     in      std_logic;
        write_i:    in      std_logic;
        addr_i:     in      std_logic_vector(1 downto 0);
        data_i:     in      std_logic_vector(31 downto 0);
        -- output
        data_o:     out     std_logic_vector(31 downto 0);
        rx_empty_o: out     std_logic;
        rx_irq_o:   out     std_logic;
        tx_irq_o:   out     std_logic;
        -- external interface
        rxd_i:      in      std_logic;
        txd_o:      out     std_logic;
        cts_i:      in      std_logic;
        rts_o:      out     std_logic
    );
end uart;
```

Internal Signals

Following are the internal signals in the UART module:

baudrate_reg	Baudrate register
hw_xonoff_ff	Hardware xon/xoff flag
tx_shift_reg	Transmitter shift register
tx_shift_en	Transmitter shift enable
tx_en	Transmitter enable
tx_rq	Transmitter request
tx_counter	Transmitter clock counter
tx_bitcnt	Transmitter bit counter
rx_shift_reg	Receiver shift register
rx_buffer_reg	Receiver buffer register
rxb_full	Receiver buffer full flag
rx_full	Receiver full flag
rx_en	Receiver enable
rx_counter	Receiver clock counter
rx_bitcnt	Receiver bit counter
rxd_ff	Receiver data flag
rts_o	Ready to send output flag
rx_empty_o	Receiver empty output flag

Read UART Registers

uart_register_file_read is an asynchronous process, by which the eP32 CPU can read the UART register at any time. When read_i=1 and ce_i=1, the register selected by addr_1 puts its contents on the data_o bus for the CPU to read.

When addr_i =0, data_o returns the baud rate count in the baud rate register. When the master clock rate is 50 MHz and the baud rate is 115,200 baud, the baud rate count is 431.

When addr_i=1, data_o returns transmitter status, where bit 8 shows Transmitter Ready state.

When addr_i=2, data_o returns receiver status, where bit 8 shows Receiver Ready, and bit 0 shows flow control state.

When addr_i=3, data_o returns the contents of the receiver buffer, where bits 0-7 show the last character just received.

```
begin

    rts_o <= hw_xonoff_ff and (not(rx_full));
    rx_empty_o <= rx_full nor rxb_full;

-- ******************************************************
--         Uart Register Circuit for Read
-- ******************************************************
  uart_register_file_read:
  process(read_i, ce_i, addr_i, baudrate_reg, tx_en, cts_i,
        hw_xonoff_ff, rxb_full, rx_buffer_reg)
    begin
    if (read_i='1' and ce_i='1') then
        case addr_i is
        when "00"   => data_o <= baudrate_reg;
        when "01"   => data_o <=   -- read TX ready flag
                "00000000" & "00000000" & "0000000" &
                ((not tx_en)and(cts_i or(not hw_xonoff_ff)))
                & "00000000";
        when "10"   => data_o <= --only cleared by rxb read
                "00000000" & "00000000" &
                "0000000" & rxb_full &
                "0000000" & hw_xonoff_ff;
        when others => data_o <= -- read&clear rxb_full flag
                "00000000" & "00000000" & "00000000" &
                rx_buffer_reg;
        end case;
    else
        data_o <= (others=>'1');
    end if;
    end process uart_register_file_read;

 -- ******************************************************
--         Uart Register File Process for Write
-- ******************************************************
  uart_register_file_write : process (rst_i, clk_i)
    begin
    if ( rst_i='1' ) then
        baudrate_reg<="00000000000000000000000110101111";
            -- 50 MHz, 115.2Kbps
        tx_shift_reg <= (others=>'0');
        tx_rq <= '0';
        hw_xonoff_ff <= '0';
    elsif (clk_i'event and clk_i='1') then
        if (tx_en='0') then
        if (write_i='1' and ce_i='1') then
            case addr_i is
            when "00"=>baudrate_reg<=data_i;
            when "01"=>
                tx_shift_reg<="11"&data_i(7 downto 0)&'0';
                tx_rq<='1';
            when "10"=>hw_xonoff_ff<=data_i(0);--flow Control
            when others => null;
            end case;
        end if;
```

Write UART Registers

uart_register_file write is a synchronous process, which writes new data into the UART registers.

When the eP32 is in the reset state, rst_i=1 also causes the UART to be reset. In the reset state, the UART initializes the baud rate register to 0x1AF (decimal 431), and sets the baud rate to 115,200 baud when the master clock is 50 MHz. In the meantime, flags tx_shift_reg, tx_rq, and hw_xonoff_ff are all cleared to 0.

Once the eP32 is in its running state, the UART responds to write commands from the CPU on the rising edge of clock clk_i when write_i=1 and ce_i=1.

When tx_en=0, the UART is not actively transmitting a character.

Writing with addr_i=0, new data is written into the baud rate register and the new baud rate will take effect immediately. One should be careful in changing the baud rate, because the external device connecting to the UART port should be set up so it responds to the new baud rate correctly.

Writing with addr_i=1, new data is written into the transmitter shift register, tx_shift_reg. The lower 8 bits of data is a character to be transmitted. Transmit request, tx_rq, is also set to start transmitting this character.

Writing with addr_i=2, the flow control bit can be changed by bit 0 of the written data.

When tx_en is not zero, the UART is transmitting a character.

If tx_shift_en=1, the rising edge of clk_i causes the character in the transmitter shift register, tx_shift_reg, to be shifted right by 1 bit. The lowest bit is shifted out to txd_o.

Transmit Process

The transmitter in the UART is running in a synchronous process, uart_tx_core.

On booting up, rst_i is set, and all registers in the UART transmitter are cleared to zero. Only txd_o is pulled up, raising the UART output line TX to high, which is the rest state of the UART output.

When transmit request, tx_rq, is set, a character is in tx_shift_reg, ready to be transmitted. tx_counter is initialized by copying the baud rate count from baudrate_reg, and the transmit bit counter, tx_bitcnt, is initialized to 11 for 1 start bit, 8 data bits and 2 stop bits. tx_en is now set to start the transmitting procedure.

```
        else
            tx_rq <= '0';
            if (tx_shift_en='1') then
                tx_shift_reg<='1'&tx_shift_reg(10 downto 1);
            end if;
        end if;
    end if;
    end process uart_register_file_write;

-- ********************************************************
--         Uart TX Core Process
-- ********************************************************
  uart_tx_core : process ( rst_i, clk_i)
    begin
    if (rst_i='1') then
        tx_counter <= (others=>'0');
        tx_bitcnt <= (others=>'0');
        txd_o <= '1';
        tx_en <= '0';
        tx_shift_en <= '0';
        tx_irq_o <= '0';
    elsif ( clk_i'event and clk_i='1' ) then
        tx_shift_en <='0';
        tx_irq_o <= '0';
        if (tx_en='0') and (tx_rq='1') and
            (cts_i='1' or hw_xonoff_ff='0') then
            tx_counter <= baudrate_reg;
            tx_bitcnt <= "1011";
            tx_en <= '1';
        elsif (tx_en='1') then
        if (tx_counter/="00000000000000000000000000000000")
            then    tx_counter <= tx_counter-1;
        elsif (tx_bitcnt/="0000") then
            tx_bitcnt <= tx_bitcnt-1;
            txd_o <= tx_shift_reg(0);
            tx_shift_en <= '1';
            tx_counter <= baudrate_reg;
        else
            txd_o <= '1';      -- mark-high=stop-bit
            tx_irq_o  <= '1';  -- transmitter empty
            tx_en<='0';    -- closed
        end if;
        end if;
    end if;
    end process uart_tx_core;
```

As tx_en is set, every rising edge causes tx_counter to be decremented. When tx_counter is 0, one bit in tx_shift_reg is shifted out to txd_o, by setting tx_shift_en, which causes the uart_register_file_write process to do the shifting. In the meantime, tx_bitcnt is decremented and tx_counter is re-initialized to baudrate_reg. This sequence is repeated 11 times to shift out all data bits in tx_shift_reg.

After all 11 bits in tx_shift_reg are shifted out, tx_en is cleared to stop the transmitting procedure. An interrupt request is activated by setting tx_irq_o. txd_o is again set to put the UART to its rest state.

Receive Process

The receiver in the UART is running in a synchronous process, uart_rx_core.

On booting up, rst_i is set, and all registers in the UART receiver are cleared to zero.

When the receiver receives a complete character, rx_full=1. On the rising edge of the master clock, the character received in rx_shift_reg is copied to rx_buffer_register, which can be sent to the eP32 when eP32 reads rx_buffer_register at location 0x80000003.

rxb_full flag is set only when rx_shift_reg is copied into rx_buffer_reg. It otherwise is always cleared to 0.

On the rising edge of every clock, the receiver input line, rxd_i, is always sampled and its state is stored into rx_ff. rxd_i is normally high when the UART is resting. When rxd_i is lowered to 0, rx_ff is cleared and it indicates that a start bit is detected and a character is coming. Activities in the next page of VHDL code cause this character to be received.

When the receiver is resting, rx_en=0. When a start bit is detected and rx_ff is cleared, the receiver is initialized to prepare receiving a new character. rx_counter is first initialized to half of the baud rate count in baudrate_reg, so that the receiver line, rxd_i, will be sampled in the middle of every bit received. rx_en is set, and rx_bitcnt is initialized to 9, for 1 start bit and 8 data bits.

When rx_en is set, every rising edge of the master clock decrements rx_counter until it is zero.

When rx_counter=0, rxd_ff is shifted into rx_shift_reg, rx_bitcnt is decremented, and rx_counter is reinitialized to the baud rate count in baudrate_reg.

When rx_bitcnt is decremented to zero, a complete character is received in rx_shift_reg. rx_full is set so that the character in rx_shift_reg will be copied into rx_buffer_reg, and be made available to the eP32. rx_irq_o is set to request an interrupt, and rx_en is cleared to receive the next character.

```
 -- ***********************************************************
--          Uart RX Core Process
-- ***********************************************************
  uart_rx_core : process ( rst_i, clk_i)
    begin
    if (rst_i='1') then
        rx_full <= '0';
        rxb_full <= '0';
        rx_irq_o <= '0';
        rx_buffer_reg <= (others=>'0');
        rx_counter <= (others=>'0');
        rx_bitcnt <= (others=>'0');
        rx_en <= '0';
        rx_shift_reg <= (others=>'0');
        rxd_ff <= '0';
    elsif ( clk_i'event and clk_i='1' ) then
        rx_irq_o <= '0';
        rxd_ff <= rxd_i;
        if (rx_full='1') then
        if (rxb_full='0') or
            (read_i='1' and ce_i='1' and addr_i="11") then
            rx_buffer_reg <= rx_shift_reg;
            rxb_full <= '1';
            rx_full <= '0';
        end if;
        else
        if (read_i='1' and ce_i='1' and addr_i="11") then
            rxb_full <= '0';
        end if;
        if (rx_en='0') and (rxd_ff='0') then
            rx_counter <= '0' & baudrate_reg(31 downto 1);
            rx_bitcnt <= "1001";
            rx_en <= '1';
        elsif (rx_en='1') then
            if(rx_counter/="00000000000000000000000000000000")
                then   -- bit-T-counting
                rx_counter <= rx_counter-1;
            elsif (rx_bitcnt/="0000") then
                -- last bit has been received
                rx_bitcnt <= rx_bitcnt-1;
                rx_shift_reg<=rxd_ff&rx_shift_reg(7 downto 1);
                rx_counter <= baudrate_reg;
                else
                    rx_irq_o <= '1';--flag for generate pulse
                    rx_full <= '1';
                    rx_en <= '0';
                end if;
            end if;
        end if;
    end if;
    end process uart_rx_core;
end behavioral;
```

5.5 GPIO Module

The VHDL code of the GPIO module is in the gpio.vhd file.

A general purpose parallel I/O port is most useful in real-time applications to interface to a wide range of external devices. In the eP32 system, such a GPIO port is included. It is designed as a 16-bit bidirectional parallel port, but the user can configure it to suit any purpose. It is declared an entity in the gpio.vhd file.

Port signals of the GPIO module are defined in the GPIO entity as follows:

Port Signal	Function
clr	Master reset
clk	Master clock
write	Write enable
read	Read enable
ce	GPIO chip select
addr	Register address
data_in	Data input from CPU
gpio_in	GPIO input
mem_conf_o	Bit0 memory select: 0-ROM; 1-RAM Bit1 CPU reset
data_out	Data output to CPU
gpio_out	Data output to GPIO output
gpio_dir	Direction select of GPIO

Registers in the GPIO module, their address and functions are as follows:

Address	Register	Function
0xE0000000	gpio_out	When written, send data to gpio port
0xE0000001	gpio_dir_reg	Select port pin direction: 0-input; 1-output
0xE0000002	gpio_in	Read gpio port

As GPIO is a module in the eP32 system, it is not connected directly to I/O pins on the eP32 system package. Therefore, gpio_in, gpio_out and gpio_dir signals are all brought out as ports in the GPIO module. These signals are used in the ep32_chip top level module to drive I/O pins.

In the eP32, a GPIO port is a 32-bit device. However, we only brought out 16 lines to pins on the LatticeXP2-5E-TN144C chip to drive 8 LEDs and to monitor 8 pushbutton switches.

Reading GPIO registers is an asynchronous process as shown in the gpio_register_file_read process. Bits in the gpio_dir register define pins as input or output. A bit set in gpio_dir makes the corresponding pin an output pin. A bit cleared in gpio_dir makes the corresponding pin an input pin. Reading the gpio_in register obtains the status of the input pins. Writing the gpio_out register sends data to the output pins.

```
- ************************************************************
-- *       General Purpose Input Output Module                *
-- *============================================================*
-- * Project:          FG in PROASIC                            *
-- * File:             gpio.vhd                                 *
-- * Author:           Chien-Chia Wu                            *
-- * 03/02/03  Chien-Chia Wu   Created.                         *
-- ************************************************************
library ieee;
use ieee.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_misc.all;
use IEEE.std_logic_unsigned.all;

entity gpio is
  port(
    -- input port
    clr:        in      std_logic;
    clk:        in      std_logic;
    write:      in      std_logic;
    read:       in      std_logic;
    ce:         in      std_logic;
    addr:       in      std_logic_vector(1 downto 0);
    data_in:    in      std_logic_vector(31 downto 0);
    gpio_in:    in      std_logic_vector(15 downto 0);
    -- output port
    data_out:   out     std_logic_vector(31 downto 0);
    gpio_out:   out     std_logic_vector(15 downto 0);
    gpio_dir:   out     std_logic_vector(15 downto 0)
  );
end gpio;
```

```
architecture behavioral of gpio is
    signal gpio_reg:   std_logic_vector(15 downto 0);
    signal gpio_dir_reg:std_logic_vector(15 downto 0);
begin
    gpio_out <= gpio_reg;
    gpio_dir <= gpio_dir_reg;

-- ********************************************************
--      GPIO Register Circuit for Read
-- ********************************************************
    gpio_register_file_read:
    process(read, ce, addr, gpio_reg, gpio_dir_reg,gpio_in)
    begin
        if (read='1' and ce='1') then
        case addr is
            when "00"  =>
                data_out<="00000000"&"00000000"& gpio_reg;
            when "01"  =>
                data_out<="00000000"&"00000000"& gpio_dir_reg;
            when others =>
                data_out<="00000000"&"00000000"& gpio_in;
        end case;
        else
            data_out <= (others=>'1');
        end if;
    end process gpio_register_file_read;
```

```
-- ********************************************************
--      GPIO Register Circuit for Write
-- ********************************************************
    gpio_register_file_write:
    process(clr, clk)
    begin
    if (clr='1') then
        gpio_reg <= (others=>'0');
        gpio_dir_reg <= (others=>'0');
    elsif ( clk'event and clk='1') then
        if (write='1' and ce='1') then
            case addr is
            when "01" => gpio_dir_reg <= data_in(15 downto 0);
            when others => gpio_reg <= data_in(15 downto 0);
            end case;
        end if;
    end if;
    end process gpio_register_file_write;
end behavioral;
```

Writing the GPIO registers is done using a synchronous process, gpio_register_file_write.

On reset, rst_i=1, and all GPIO registers are cleared to zero.

When running, on the rising edge of clk_i, if ce_i= and write=1, data from the CPU on the data_i bus are written into the register selected by addr_i. Writing to the gpio_reg register send data to output pins. Writing to the gpio_dir register defines the input and output pins.

5.6 Remarks

Here I had just shown you the design of a complete 32-bit microprocessor in VHDL. What I want to convey is the idea that CPU is not difficult. It can be very simple. It was made very complicated because CPU designers did not fully understand the fundamental components necessary for a CPU to function, and thus made designs unnecessarily complicated.

I cannot overemphasis the fact that the eP32 CPU executes all instructions in a single clock cycle. All prior CPU designs required many clock cycles to execute an instruction. Designers tried very hard to cover up this deficiency with pipelining and other techniques, and made the CPU even more complicated.

This design of eP32 microprocessor is only a starting point for you to design and build your own microprocessor. You should consider extending this design in the following directions:

For immediate applications, you should consider adding new I/O modules to handle specific tasks in your applications. I gave you a GPIO and a UART as examples. You can incorporate existing I/O modules into your design. If you understand your

tasks, it is probably easier to design your own I/O modules than pulling ‘library modules’ off the shelf.

For long term development, you should consider adding new instructions to the CPU core. I am sure you feel constrained by the very small instruction set I put into the eP32 CPU. There are spaces for 37 more instructions in the current eP32 architecture. If you are ambitious, why not encode instructions in bytes? Then, you can have 256 instructions. Now, you are at a point to implement a Java Virtual Machine with byte codes.

The possibility is only limited by your imagination.

How about software? If one changed hardware design, who's going to provide software to make use of improvments?

As President Obama said: “Yes, we can!”

Read the next chapter.

Chapter 6. Metacompilation of the eP32

In 1990, I hosted monthly meetings of the Forth Interest Group. The morning sessions were generally for FORML, Forth Modification Laboratory, where we discussed how to enhance the FORTH of the time. We were brain-storming what FORTH would be like in the next century. Two different directions were debated. Tom Zimmer and Andrew McKewan wanted a FORTH for Windows, and developed Win32Forth to take advantage of the popular Windows platform. It became a huge and complicated system. Bill Muench and I wanted a simple FORTH portable to all new and exciting microprocessors coming in the future. We developed eForth and it was implemented on 30 some different microprocessors and microcontrollers by many volunteers.

In the meantime, I also worked with Chuck Moore on his next FORTH chip, the MuP21. It was targeted to a 1.2 micron CMOS process available from Orbit Semiconductor on shared 5 inch wafers. Dies were 2.4x2.4 mm, and it forced Chuck to strip bare his CPU. He reduced instructions to 25, and fit a 20-bit microprocessor on this small die, with an NTSC video coprocessor and a DRAM memory coprocessor. It was a marvelous design, but we ran out of money before it was perfected.

I compared the designs of eForth and the MuP21, and found great similarity, in spite of the completely different origins of these two designs. eForth is a software design and the MuP21 is a hardware design. However, they both were based on primitive instruction sets with about 30 instructions. Many instructions were identical in these two instruction sets. Those instructions which were different, were different because of hardware constraints. I was able to implement eForth on the MuP21, and it was a very pleasant system, a real FORTH language on a real FORTH CPU.

After the MuP21, Chuck and I went our separate ways. He founded iTV and Intellesys, and built multiprocessor chips based on the MuP21 core design. I discovered FPGAs, and developed scalable P-series microprocessors based on the same core, implementing 16-, 24- and 32-bit versions of the P-microprocessors.

A young fellow in Taiwan, Mr. Cheah-shen Yap, ported eForth to Windows to become the weForth system. He further enhanced it and released it as the F# system. These are the simplest FORTH implementations for Windows, but they can call all Windows APIs to build applications running on a PC. I used both to write metacompilers for embedded systems. However, for the eP32, I preferred weForth, because it has a simpler user interface to load applications. When weForth.exe is executed on Windows, it loads a start.f file, which loads in Windows utilities and application files. F# has a more sophisticated graphical user interface, and gives the user better ways to organize software projects. For an eP32 metacompiler, however, weForth is more than enough, and it is easier to document and to explain.

The complete command set of weForth is shown in Appendix B for your reference.

My goal is to build a FORTH microprocessor based on the eP32 CPU on an FPGA chip, the LatticeXP2-5E, hosted on a LatticeXP2 Brevia Development Kit. FPGA synthesis and programming tools are provided in the ispLEVER Development system supplied by Lattice. The FORTH system on the eP32 is an eForth system, and I build this eForth target system in weForth, an eForth system running on a Windows PC.

In FORTH terminology, a metacompiler is a FORTH program which produces an image of program memory, which is copied into the memory of a target microprocessor. When the target microprocessor powers up, a FORTH system is booted up to interact with its user.

I believe the best way to explain this eForth system is through the source code of the eForth metacompiler in weForth that produces this system. I like to take the same approach in presenting the eP32 hardware by commenting on its VHDL source files. I will put eForth source code on left pages, and commentary on opposing right pages. Going through source code almost line by line, I hope that I can make clear the process of producing a target system on the eP32, as well as make clear the code and other relevant information that go into program memory in the eP32.

Before going through source code files in the eForth metacompiler, I will first show you the metacompiling process in weForth, and how an eP32 target image is generated. In addition, I will show you a simulator in weForth, which simulates the eP32 eForth as an eP32 running on a Brevia Development Kit. This way you can try running an interactive FORTH system on a simulated eP32 without the Brevia Kit. It is a good way to learn how FORTH works. You have two FORTH systems to experiment with: weForth as a Windows application, and eP32 eForth as an embedded application on the Brevia Kit.

6.1 Metacompiling the eP32

All source code of the eP32 eForth system is contained in the ep321_xp2.zip file. weForth and its Windows utilities are also included here.

Unzip file ep32q_xp2.zip and put all the files into a folder named “ep32q_xp2”. Start weForth by double clicking weforth2.exe in the ep32q_xp2 folder, as shown in Figure 29.

weForth opens a console window, loads the eForth metacompiler and generates a new eP32 target system.

A memory image of the eP32 eForth target system is stored in file mem.mif. While building this system, weForth prints out large amount of messages on its console window.The console window at the end of the metacompilation process is shown in Figure 30.:

Scroll the console window back to its beginning, and you can see that weForth loads several system files, win32.f, api.f, and ui.f, to bring in the necessary Windows APIs, as shown in Figure 31.

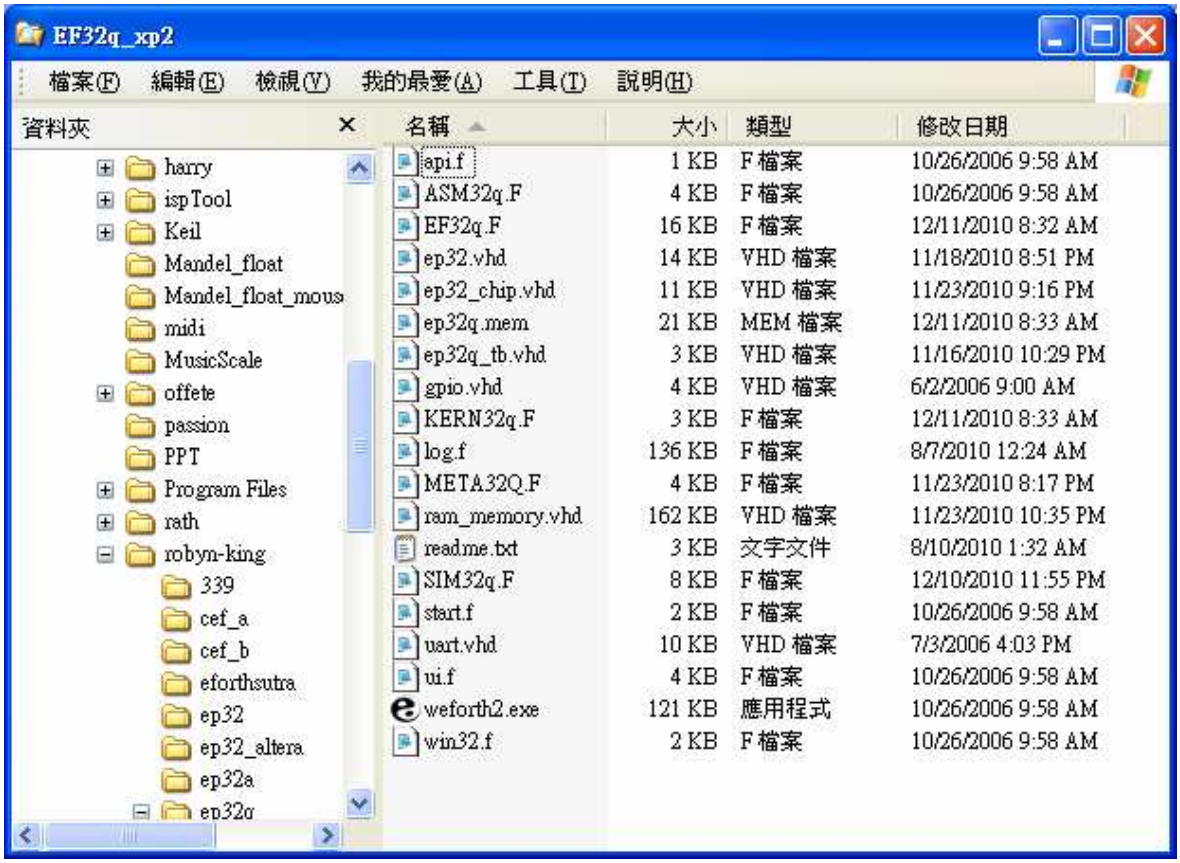

Figure 29. ep32 Project Folder

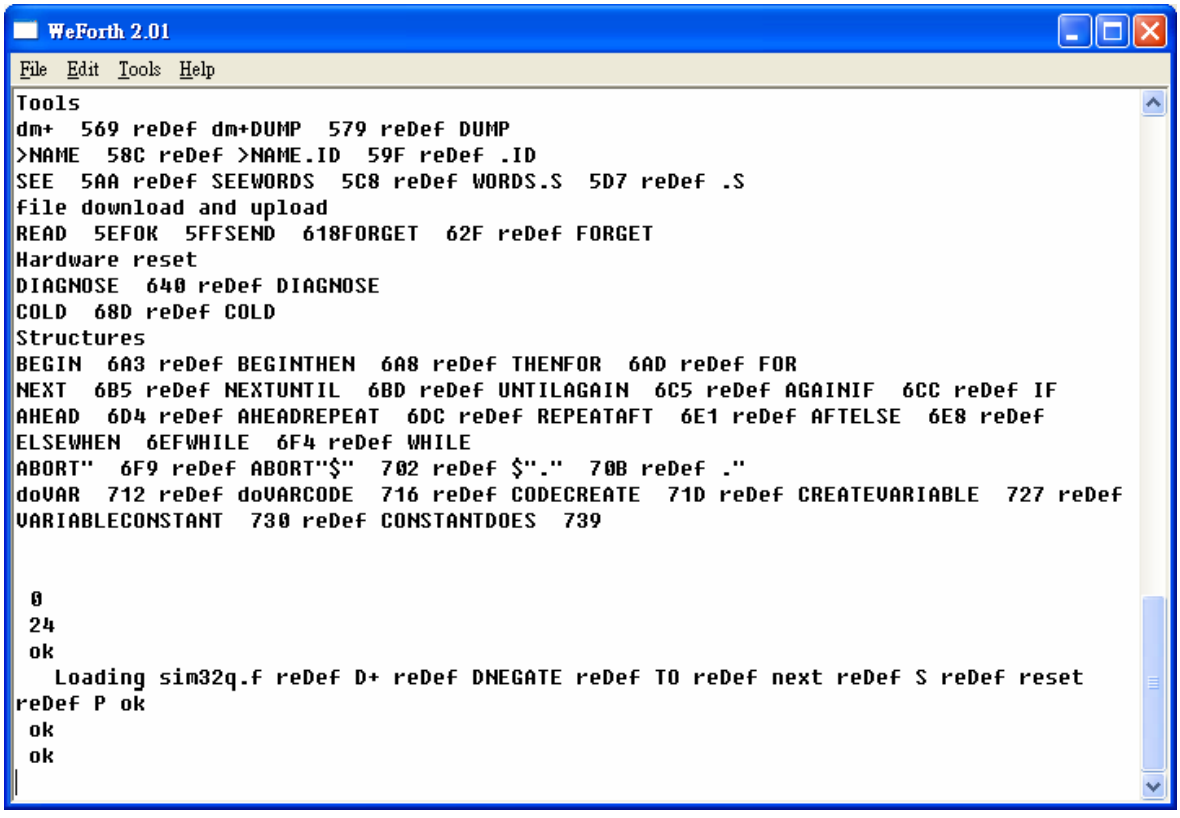

Figure 30. Bootup ep32 Metacompiler

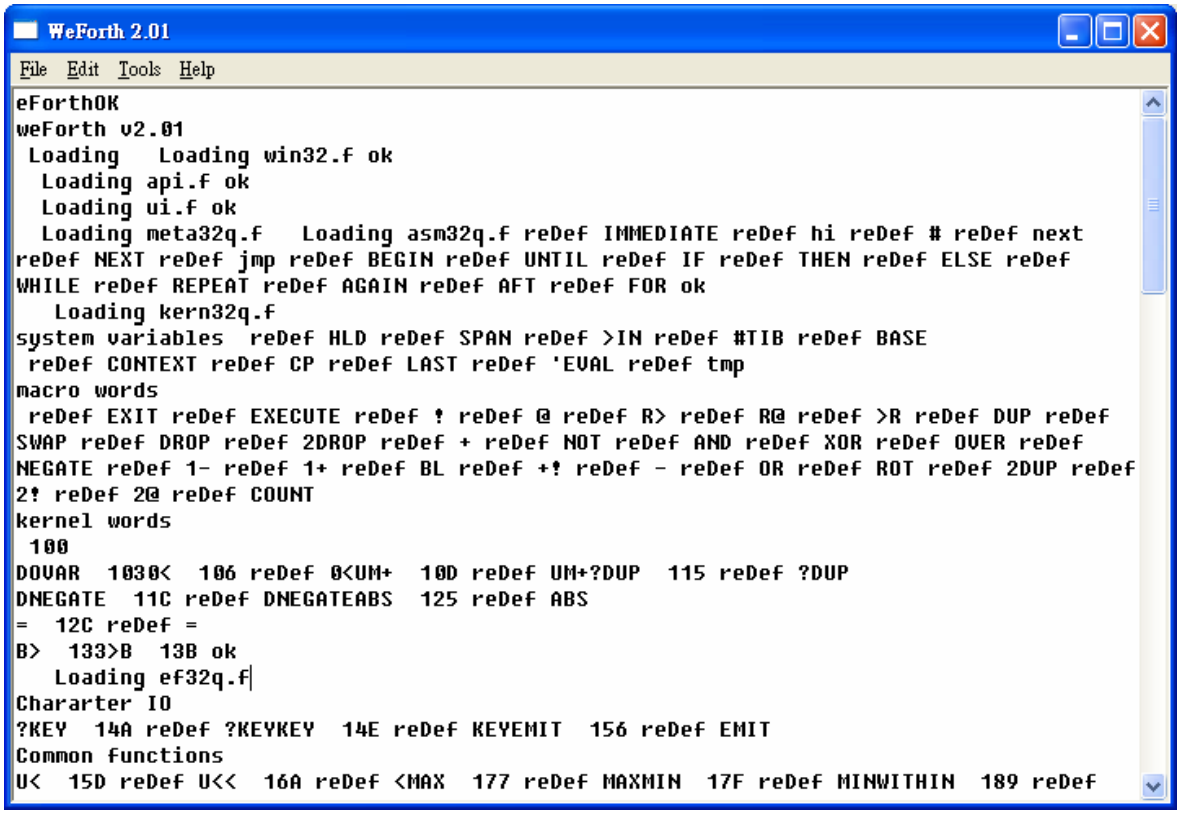

Figure 31. Beginning of Metacompilation

The next file loaded is meta32q.f, which is the eP32 metacompiler. It first loads asm32q.f to bring in the eP32 assembler. It prints out a list of command names followed by a “reDef” message. These commands are defined in the eP32 assembler, preparing to assemble commands in the eP32 kernel.

The next file loaded is kern32q.f, which first defines many macro commands. Then it starts building the eP32 kernel at target memory location $100. There you can see names of target commands followed by their code field addresses. They form a symbol table, which you can use to look up names and addresses of target commands.

After the kernel is built, the metacompiler loads in ef32q.f, which compiles the complete eForth target system, and writes its FORTH dictionary out into a file mem.mif. This file is used to initialize the RAM_DQ memory array in the ram_memory.vhd file, and to synthesize the eP16 microcontroller in the FPGA chip on the Brevia2 Kit as mentioned in the last section.

After the eP32 target system is built, the metacompiler loads sim32q.f, which is an eP32 simulator. This simulator executes eP32 instructions compiled by the metacompiler, and can faithfully simulate the eP32, instruction by instruction.

Simulating the eP32

Once the sim32q.f simulator is loaded, type the command:

```
HELP
```

and a list of simulator commands appear, as shown in Figure 32.

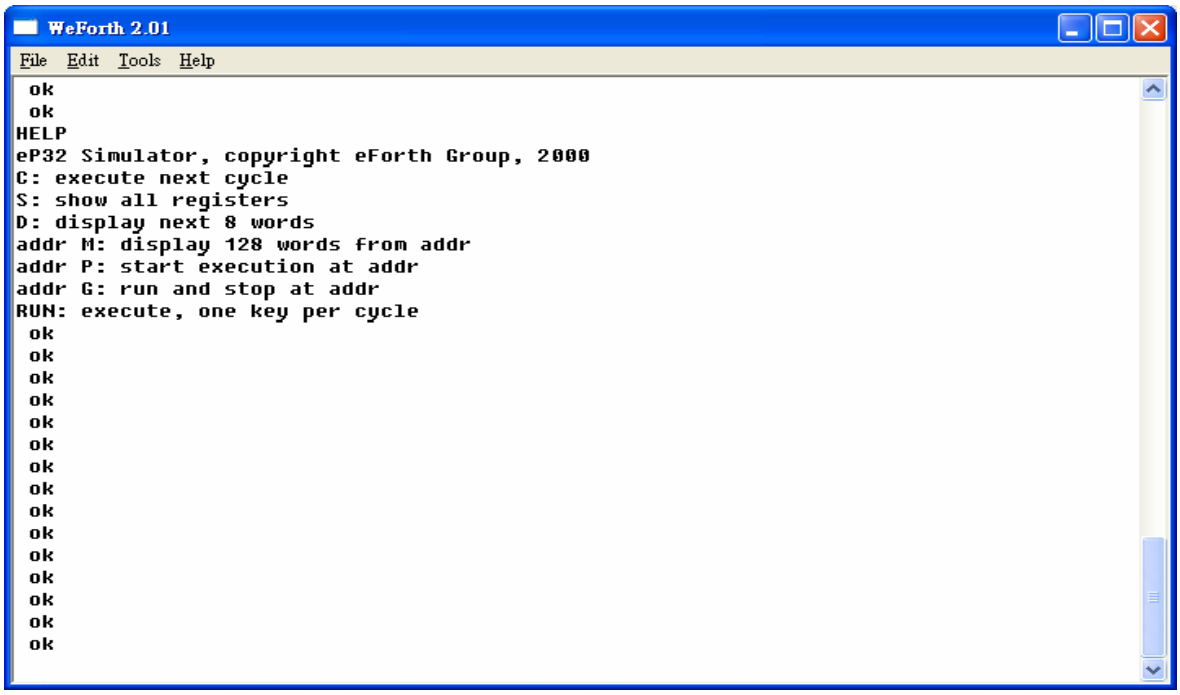

Figure 32. HELP Directions of eP16 Simulator

Type this command:

```
-1 G
```

and the simulator boots up the eP32 eForth system and prints out its sign-on message:

```
eP32q v2.05
```

This is what you see next in Figure 33.

Now you can exercise eP32 eForth by typing in FORTH commands.
The following screen shot shows results when you type command:

```
WORDS
```

If you care to count them, there are 167 commands. These commands are documented in Appendix B.

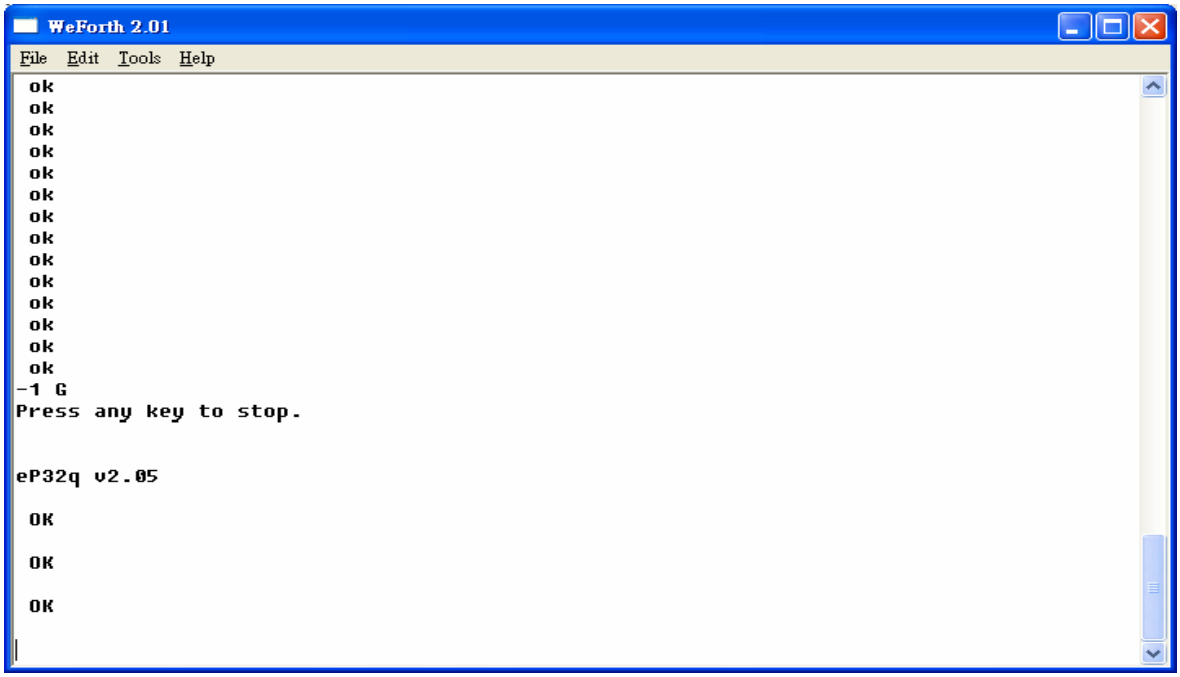

Figure 33. eP16 in Simulation

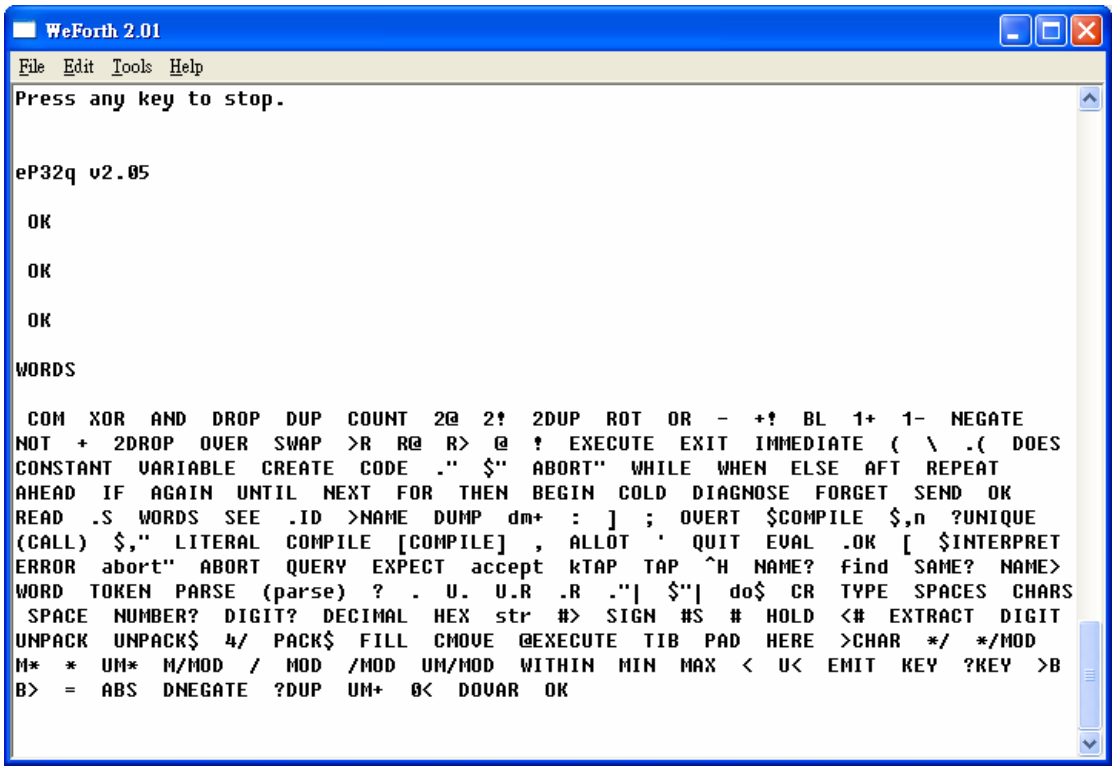

Figure 34. WORDS in eP32

Here are more eForth commands you can type into the weForth console to test the eForth system:

```
HEX 0 80 DUMP
SEE WORDS
```

```
HERE .
1 2 + .
: TEST1 1 2 3 4 5 ;
TEST1
.S
: TEST2 10 FOR R@ . NEXT ;
TEST2
: TEST3 IF 1 ELSE 2 THEN . ;
0 TEST3
1 TEST3
: TEST4 CR ." HELLO, WORLD!" ;
TEST4
```

After these tests, the weForth console looks as follows.

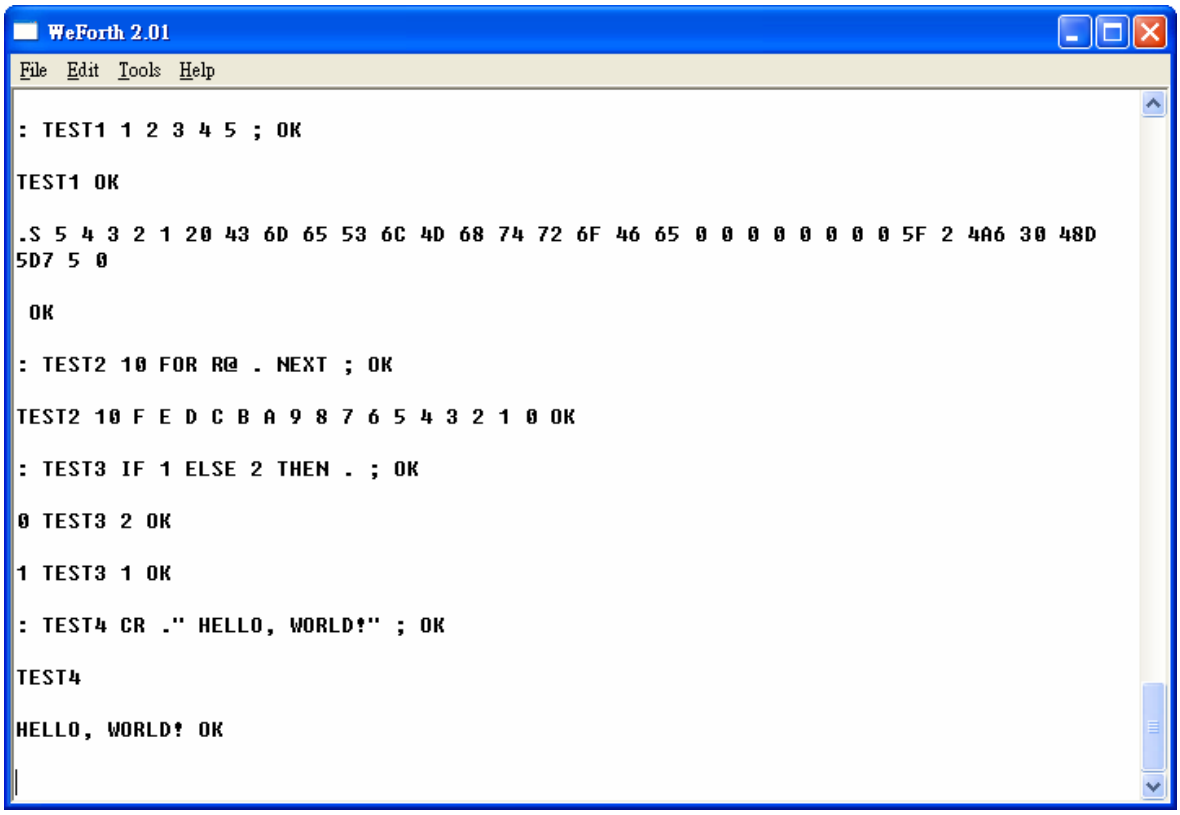

Figure 35. Tests of eP32 Simulator

6.2 The eP32 Metacompiler

The eP32 metacompiler is contained in file meta32q.f.

"Metacompiler" is a term used by a FORTH programmer to describe the process of building a new FORTH system on an existing FORTH system. The new FORTH system may run on the same platform as the old FORTH system. It may be targeted to a new platform, or to a new CPU. The new FORTH system may share a large portion of FORTH code with the old system, hence the term "metacompilation". In a sense, the metacompiler is very similar to a conventional cross assembler/compiler.

start.f is similar to a MAKE file in UNIX. FORTH commands in this file are executed by the weForth system upon startup. It loads in a metacompiler in meta32q.f, which compiles a target eForth system for the eP32. It produces a memory image file, which will be used to initialize memory blocks by IPexpress in the Lattice ispLEVER system to program the LatticeXP2-5E FPGA chip. meta32q.f contains the following commands to load source code from many other files:

asm32q.F	eP32 assembler
kern32q.F	Primitive commands in eP32 eForth
ef32q.F	Compound commands in eP32 eForth
sim32q.F	eP32 simulator

```
( meta32.f for weforth )

HEX
VARIABLE debugging?

: .head ( addr -- addr )
   >IN @ 20 WORD COUNT TYPE SPACE >IN !
   DUP .
   ;
: cr CR
   debugging? @
   IF .S KEY 0D = ABORT" DONE"
   THEN
   ;

: forth_' ' ;
: forth_dup DUP ;
: forth_drop DROP ;
: forth_over OVER ;
: forth_swap SWAP ;
: forth_@ @ ;
: forth_! ! ;
: forth_and AND ;
: forth_+ + ;
: forth_- - ;
: forth_word WORD ;
: forth_words WORDS ;
: forth_.s .S ;
: CRR cr ;
: forth_.( [COMPILE] .( ;
: forth_count COUNT ;
: forth_r> R> ;
: -or   XOR ;
: >body 5 + ;
: forth_forget FORGET ;

CREATE ram  8000 ALLOT
: reset   ram 8000 0 FILL ;
: ram@   4 * ram +  @ ;
: ram!   4 * ram +  ! ;
: binary 2 BASE ! ;
: four   3 FOR DUP ram@ 9 U.R  1+ NEXT ;
: show ( a)   0F FOR CR  DUP 9 .R SPACE
      four 2 SPACES four NEXT ;
: showram 0 0B FOR show NEXT DROP ;

: dump-ram
   BASE @ binary 0
   1000 FOR AFT
      CR DUP ram@ <# 1F FOR # NEXT #> TYPE
      1+
   THEN NEXT
   DROP BASE ! CR
   ;
```

We start here to discuss metacompiler commands in the meta32q.f file. All other files referred to in this file will be discussed in their separate sections.

debugging?	A variable containing a switch to turn break points on and off. When debugging? is set to -1, compilation will stop and the data stack is dumped when a “cr” command is executed. Sprinkling “cr” commands in the source code file allows you to watch the progress of metacompilation and even stops it when necessary.
.head	Display name of a command that is about to be compiled. It is used to display a symbol table. You can look up the code field address of any command in this table.
cr	Stop metacompilation if debugging? is -1, and dump data stack. If you press control-A, metacompilation is aborted. Otherwise, metacompilation continues. It is a NOP if debugging? is 0.

During metacompilation, FORTH commands will be redefined so that they compile subroutine call instructions or assemble other machine instructions into the target memory image. There are numerous occasions where the original behavior of a FORTH command must be exercised. To preserve the original behavior of a FORTH command, it is assigned a different name. Thereby after a command is redefined, we can still exercise its original behavior by invoking the alternate name.

For example, “+” is a FORTH command that adds the top two numbers on the data stack in the weForth system. Then in the kern32q.f file, a new “+” command is defined to assemble an ADD instruction in the target eP32 system. If you still need to add two numbers, you must use the alternate command “forth_+” as shown below. All the weForth commands you need to use later must be redefined as “forth_xxx” commands. If you neglect to redefine them, you will find that the system behaves very strangely.

The eP32 executes program words and accesses data in the memory range 0-1FFF. In weForth we allocate a 32k byte memory array, “ram”, to hold the eP32 target image. This array contains code and data to be copied into eP32 internal memory at 0, to be executed on the eP32 chip.

ram	Memory array in weForth for the eP32 target image. It has a logical base address of 0 for the eP32. Code and data words in the target are stored in this array.
ram@	Replace a logical address on stack with data stored in “ram” image array.
ram!	Store second integer on stack into logical address of “ram” image array.
reset	Clear “ram” image array, preparing it to receive code and data for the eP32.
four	Display four consecutive words in target.
show	Display 128 words in target from address “a”. It also returns a+128 to “show” the next block of 128 words.
showram	Display the entire eP32 image of 2k words.
dump-ram	Display 4k words of data in binary.

```
VARIABLE hFile
CREATE CRLF-ARRAY 0D C, 0A C,

: CRLF
     hFile @
     CRLF-ARRAY 2
     PAD ( lpWrittenBytes )
     0 ( lpOverlapped )
     WriteFile
     IF ELSE ." write error"
     QUIT THEN
   ;

: open-mif-file
   Z" mem.mif"
   $40000000 ( GENERIC_WRITE )
   0 ( share mode )
   0 ( security attribute )
   2 ( CREATE_ALWAYS )
   $80 ( FILE_ATTRIBUTE_NORMAL )
   0 ( hTemplateFile )
   CreateFileA hFile !
   ;

: write-mif-line
     PAD ( lpWrittenBytes )
     0 ( lpOverlapped )
     WriteFile
     IF ELSE ." write error" QUIT THEN
   CRLF
   ;

: write-mif-header
     hFile @
     $" #Format=AddrHex "
     write-mif-line
     hFile @
     $" #Depth=4096 "
     write-mif-line
     hFile @
     $" #Width=32 "
     write-mif-line
     hFile @
     $" #AddrRadix=3 "
     write-mif-line
     hFile @
     $" #DataRadix=3 "
     write-mif-line
   ;
```

The eP32 metacompiler builds a target image for the eP32 chip in “ram”, a memory array in weForth. This image eventually will be imported to the ispLEVER system so that this target image will be incorporated in the RAM_Q module, which will be synthesized with the eP32 core logic to be implemented in the LatticeXP2-5E FPGA chip. IspLEVER requires that the target image be written in a file conforming to its Addressed-Hex format, which consists of a header with a few lines of system information in ASCII text, and then a body containing memory information in hexadecimal numbers. The header and first few lines of the body are as follows:

```
#Format=AddrHex
#Depth=4096
#Width=32
#AddrRadix=3
#DataRadix=3
#Data
0:68D
24:80
25:A
26:7C6
27:7C8
28:7C6
29:4A0
2A:4D2
2D:7C6
101:564F4405
102:5241
103:1805E79E
104:101
105:3C3002
106:1179E79E
107:3000109
108:1A69405E
109:A05E79E
```

In the body of mem.mif, each line of data consists of an address and its contents as hexadecimal numbers separated by a colon character.

hFile	A variable holding a file handle.
CRLF	Insert a carriage return and a line feed into the currently opened file.
open-mif-file	Open a file named mem.mif for writing.
write-mif-line	Write one line of text into current file.
write-mif-header	Write a header required by ispLEVER into current file.

“mif” is a leftover term used when I was implementing the eP32 for the Xilinx FPGA, and its development system expected a memory file to be in its mif format. Now, ispLEVER from Lattice wants a mem file. So be it. FPGA development systems from Actel and Altera also require different memory file formats. It is easy to conform to their requirements by changing these xxx-mif-yyy commands here.

```
: write-mif-data
   0 ( initial ram location )
   $1000 FOR AFT
    DUP ram@ IF
      hFile @
      OVER
      <# 3A HOLD #S #>
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
      hFile @
      OVER ram@
      <# #S #>
      PAD ( lpWrittenBytes )
      0 ( lpOverlapped )
      WriteFile
      IF ELSE ." write error" QUIT THEN
      CRLF
    THEN
      1+
   THEN NEXT
   DROP ( discard ram location )
   ;

: close-mif-file
   hFile @ CloseHandle DROP
   ;

: write-mif-file
   open-mif-file
   write-mif-header
   write-mif-data
   close-mif-file
   ;

FLOAD asm32q.f
FLOAD kern32q.f
FLOAD ef32q.f
write-mif-file
FLOAD sim32q.f
```

write-mif-data	Write a 4k word image of the eForth System from memory array “ram” to the mem.mif file.
close-mif-file	Close the mem.mif file.
write-mif-file	Write a file mem.mif containing 2k words of the eForth System according to the Address-Hex format required by IPexpress.

IPexpress in the ispLEVER FPGA development system expects an eP32 target image in Hex-Address format. A mem file has a header containing system information, and a body that contains memory data in hexadecimal ASCII characters.

Write-mif-file opens an mem.mif file, writes a header, writes data, and then closes the file. The mem.mif file must be copied into the eP32 project in the ispLEVER system to be synthesized with the eP32 VHDL files, in order to build the eP32 system for the LatticeXP2-5E FPGA chip.

The eP32 metacompiler continues to load the eP32 assembler in asm32q.f, the eP32 kernel in kern32q.f, and the eForth system in ef32q.f with the following commands:

```
FLOAD asm32q.f
FLOAD kern32q.f
FLOAD ef32q.f
```

The target image is complete, and can be now written out into mem.mif by the write-mif-file command.

The metacompiler now loads in the simulator in sim32q.f with:

```
FLOAD sim32q.f
```

The eP32 eForth system can now be simulated in weForth. It is most satisfying to see that the output of this simulator matches exactly what is produced by the eP32 eForth system in the XP2 FPGA chip. This simulator is a simulator, working at machine instruction level. It is much more convenient to run than the Active-HDL simulator which works at clock cycle level. Once a development cycle is closed in this fashion, we have very high confidence that any software change in source code of the eForth system will work in the FPGA, if it first passed this high-level simulator.

6.3 The eP32 Optimizing Assembler

The ASM32q.f file contains a structured, optimizing assembler for the eP32. It packs up to 5 machine instructions into one 32-bit program word. The strategy of this eP32 assembler is to clear a program location pointed to by a variable “hw”, preparing it to receive up to 5 machine instructions. Assembly commands are executed to insert machine instructions into consecutive slots. Assembly commands make necessary decisions as to whether to add more instructions to the current program word, or start a new program word.

The eP32 has two types of instructions, 32-bit long instructions and 6-bit short instructions. The long instruction format is:

31-30	29---24	23---18	17---12	11----6	5-----0
00	cccccc	aaaaaa	aaaaaa	aaaaaa	aaaaaa

and the short instruction format is:

31-30	29---24	23---18	17---12	11----6	5-----0
00	cccccc	cccccc	cccccc	cccccc	cccccc

cccccc is a 6-bit machine instruction, and aaaaaa-aaaaaa-aaaaaa-aaaaaa is a 24-bit address. Each 32-bit program word can contain a long instruction, or 5 short instructions.

Assembly commands for long instructions are defined by the word JUMP, and assembly commands for short instructions are defined by the word INST. Defining words in FORTH makes this optimizing assembler very simple and very efficient.

However, this assembler does not use long instructions directly to redirect program flow. Instead, it uses standard FORTH control structure commands to build control structures in assembly programs. It thus avoids complications in labels and forward referencing. It significantly simplifies this optimizing assembler.

The eP32 eForth system is based on the Subroutine Threading Model, in which a compound command consists of a list of subroutine call instructions. As call and return instructions execute in a single cycle, the eP32 is very efficient in executing FORTH compound commands as a list of subroutine call instructions. Compound commands in the form of lists of subroutine call instructions can be freely intermixed with other machine instructions. Thus this optimizing assembler becomes an optimizing compiler as well.

```
HEX

VARIABLE h
VARIABLE lasth 0 lasth !         \ init linkfield address lfa

: namer! ( d -- )
   h @ ram!                      \ store double to code buffer
   1 h +!                        \ bump nameh
   ;

: COMPILE-ONLY 40 lasth @ ram@ XOR lasth @ ram! ;
: IMMEDIATE   80 lasth @ ram@ XOR lasth @ ram! ;

VARIABLE hi
VARIABLE hw
VARIABLE bi ( for byte packing)
: align  14 hi ! ;
: org  DUP . CR h !  align ;
: allot ( n -- ) h +! ;

CREATE mask  3F000000 , FC0000 ,  3F000 ,  FC0 ,  3F ,
: #,   ( d ) h @ ram!  1 h +! ;
: ,w   ( d ) hw @ ram@  OR  hw @ ram! ;
: ,i   ( d ) hi @ 14 =  IF  0 hi !  h @ hw !  0 #,  THEN
           hi @ mask + @ AND  ,w  4 hi +! ;
: spread ( n - d ) DUP 40 * DUP 40 * DUP 40 * DUP 40 * + + + + ;
: ,l   ( n ) spread ,i ;
: ,b   ( c ) bi @ 0 = IF 1 bi ! h @ hw ! 0 #, ,w EXIT THEN
           bi @ 1 = IF 2 bi ! 100 * ,w EXIT THEN
           bi @ 2 = IF 3 bi ! 10000 * ,w EXIT THEN
           0 bi ! 1000000 * ,w ;

: inst CONSTANT DOES> R> @ ,i ;
1E spread inst nop

: anew BEGIN hi @ 14 < WHILE nop REPEAT 0 bi ! ;
: # ( d )  0A spread ,i  #, ;
: ldi # ;
: LIT ( d -- ) # ;
```

COMPILE-ONLY	Patch Bit 6 in first word of name field in current target command. Text interpreter checks it to avoid executing compiler commands.
IMMEDIATE	Patch Bit 7 in first word of name field in current target command. Compiler checks it to execute commands while compiling.

h	A variable pointing to the next free memory cell at the top of the target dictionary.
lasth	A variable pointing to the name field of the current target command under construction.
namer!	Compile a 32-bit value, “d”, to the top of the target dictionary.
hw	A variable pointing to a new program word being constructed.
hi	A variable pointing to a slot to pack the next machine instruction.
bi	A variable pointing to a byte to pack the next ASCII character.
align	Initialize pointer “hi” to start assembling a new program word.
org	Initialize pointer “h” to a new address to start assembling.
allot	Add a “n” to pointer “h”. It skips an area in target memory and starts assembling above this area.
mask	An array of 5 masks to isolate one 6-bit machine instruction from a 32-bit instruction pattern. A machine instruction can be assembled in one of 5 instruction slots selected by “hi”.
#,	Compile “d” to top of target dictionary. It is the most primitive assembler and compiler. The eP32 assembler is an extension of this primitive assembly command.
,w	OR “d” to the program word pointed to by “hw”. It generally fills the address field in the current program word.
spread	Repeat 6-bit machine instruction “n” in all 5 slots to form a 32-bit instruction pattern. “mask” uses it to select a slot for assembling.
,i	Use “hi” to select one machine instruction in “d” and assemble it into the program word selected by “hw”.
,l	Spread a 6-bit machine instruction to a 32-bit pattern and assemble a machine instruction with “,i”.
,b	Pack byte “b” into current program word. Pointer “bi” determines which byte field to pack. “bi” is incremented to facilitate packing of next byte.
inst	Define short instruction assembly commands. It creates a short instruction assembly command like a constant. When a short instruction assembly command is later executed, this constant is retrieved as an instruction pattern and a short machine instruction is assembled into the current program word by command “,i”.
nop	First short instruction assembly command defined by “inst”.
anew	Fill current program word with NOPs and initialize hi and hw to assemble new machine instructions in the next program word.
#	Assemble a load literal LDI instruction. Its literal value is assembled in the next program word pointed to by "h".
ldi	Alias of “#”.
LIT	Alias of “#”.

```
: (makehead)
   anew
   20 WORD                          \ get name of new definition
   lasth @ namer!                   \ fill link field of last word
   h @ lasth !                      \ save nfa in lasth
   COUNT DUP ,b                     \ store count
   1- FOR
      COUNT ,b                      \ fill name field
   NEXT
   DROP anew
   ;

: makehead
   >IN @ >R                         \ save interpreter pointer
   (makehead)
   R> >IN !                         \ restore word pointer
   ;

: $LIT ( -- )
   anew
   22 WORD
   COUNT DUP ,b ( compile count )
   1- FOR
      COUNT ,b ( compile characters )
   NEXT
   DROP anew ;

: jump CONSTANT DOES> anew R> FFFFFF AND @ OR #, ;
     0 jump bra            0 jump jmp
2000000 jump bz
3000000 jump bc
4000000 jump call
5000000 jump next
5000000 jump NEXT
5000000 jump <NEXT>

: return CONSTANT DOES> R> @ ,i anew ;
 1 spread return ret
 6 spread return times
```

In the eP32 eForth system, all target commands are compiled in a target dictionary, and linked as a list. Each target command has a link field of one 32-bit word, a variable length name field in which the first byte contains a length followed by the ASCII code of the name string, null filled to a 32-bit word boundary, and a variable-length code field containing 32-bit program or data words. Primitive target commands have machine instructions in their code fields. Compound target commands generally have call instructions in their code fields. As call instructions can intermix with other machine instructions, primitive words are indistinguishable from compound words.

(makehead)	Build a header for a new target command. The header includes a link field and a name field. The address of the name field in the last target command is stored in “lasth”, and is compiled into the link field. “h” points to the name field of the new command, and is copied into “lasth”. Now, the following string is packed into the name field, starting with its length byte, and null filled to the word boundary. Now, “h” points to the code field of this new target command.
makehead	Build a header with (makehead) and save the name string to define a compiler command in metacompiler. It displays the name and code field address. A string can be used repeatedly by saving and restoring its pointer in a “>IN” word.
$LIT	Compile a packed string for a string literal. It works similarly as (makehead). However, the name string is delimited by the space character (ASCII 0x20), while a string literal is delimited by the double-quote character (ASCII 0x22).
jump	A defining command that creates long instruction assembly commands. It uses transfer instruction code like a constant. When a long instruction assembly command is later executed, it retrieves this code, ORs it with a 24 bit address, and assembles a transfer instruction in the target dictionary.

Following are the eP32 long instruction assembly commands defined by “jump”:

bra	Assemble a branch always instruction, BRA.
bz	Assemble a branch on zero instruction, BZ.
bc	Assemble a branch on carry instruction, BC.
call	Assemble a subroutine CALL instruction.
next	Assemble a loop NEXT instruction.

return	A defining command to create assembly commands that abandon remaining slots in the current program word, and start fetching the next program word.
ret	Assembly command to return from subroutine call. “ret” is similar to “nop”, in that all machine instructions following them in the same program word will be ignored.
times	Assembly command to terminate a micro loop. It is not implemented in eP32.

```
: begin anew h @ ;
: until bz ;
: untilnc bc ;
: jmp  bra ;

: if     h @ 0 bz ;  ( 5F80000 )
: ifnc   h @ 0 bc ; ( 5F40000 )
: skip   h @ 0 bra ;       ( 5FC0000 )
: then   begin OVER ram@ OR SWAP ram! ;
: else   skip  SWAP then ;
: while  if SWAP ;
: whilenc ifnc  SWAP ;
: repeat  bra then ;
: again   bra ;
: aft ( a -- a' a" )
   DROP skip begin SWAP ;

: BEGIN anew h @ ;
: UNTIL bz ;
: UNTILNC bc ;
: JMP  bra ;

: IF     h @ 0 bz ;  ( 5F80000 )
: IFNC   h @ 0 bc ; ( 5F40000 )
: SKIP   h @ 0 bra ;       ( 5FC0000 )
: THEN   begin OVER ram@ OR SWAP ram! ;
: ELSE   skip  SWAP then ;
: WHILE  if SWAP ;
: WHILENC ifnc  SWAP ;
: REPEAT  bra then ;
: AGAIN   bra ;
: AFT ( a -- a' a" )
   DROP skip begin SWAP ;

: ':  begin  .head CONSTANT  DOES> R> @  call ;
: CODE  makehead ': ;          \ for eforth kernel words
: code  makehead ': ;          \ for eforth kernel words

  08 spread inst ldrp  09 spread inst ldxp
( 0A spread inst ldi) 0B spread inst ldx
  0C spread inst strp  0D spread inst stxp
  0E spread inst rr8   0F spread inst stx
  10 spread inst com   11 spread inst shl
  12 spread inst shr   13 spread inst mul
  14 spread inst xor   15 spread inst and
  16 spread inst div   17 spread inst add
  18 spread inst popr  19 spread inst xt
  1A spread inst pushs 1B spread inst over
  1C spread inst pushr 1D spread inst tx
( 1E spread inst nop ) 1F spread inst pops

: for ( -- a )
   pushr begin ;
: FOR ( -- a )
   pushr begin ;
```

The eP32 transfer instructions are not used directly. They are used by control structure commands to construct control structures. These commands are in lower case for the assembler and in upper case for the compiler:

Command	Function
begin	Mark current location in target for later address resolution.
until	Terminate a begin-until loop if zero-flag is cleared.
untilz	Terminate a begin-until loop if zero-flag is set.
untilnc	Terminate a begin-until loop if carry-flag is cleared.
jmp	Jump to the address on top of the data stack.
if	Start a conditional branch structure. Assemble a bz instruction.
ifnc	Start a conditional branch structure. Assemble a bc instruction.
skip	Start a branch structure. Assemble a bra instruction.
then	Terminate a conditional branch structure by resolving the branch instruction at “if” or “else”.
else	Resolve branch instruction at “if”, and start a branch structure. Assemble a bra instruction.
while	Start a conditional branch structure in a begin-while-repeat loop. Assemble a bz instruction.
whilenc	Start a conditional branch structure in a begin-while-repeat loop. Assemble a bc instruction.
repeat	Terminate a begin-while-repeat loop, and assemble a bra instruction to “begin”.
again	Terminate a begin-again loop, and assemble a bra instruction to “begin”.

CODE defines new primitive commands in the eP32 target. Primitive commands thus defined will assemble CALL instructions in code fields of compound commands in the eP32 target. Using the Subroutine Threading Model, primitive commands are the same as compound commands. Their difference is only conceptual.

‘:	Define a nameless subroutine. “begin” points to the code field and is defined as a constant in the metacompiler. The run time behavior of this constant is changed to execute commands after DOES>, which uses the saved code field address to assemble a CALL instruction. It also displays the name of the new command and its execution address on the terminal, with the .head command.
CODE	Define a new target command. It creates a new header in the target, and then uses ‘: to start a new subroutine. It also creates an assembly command in the metacompiler. This assembly command assembles a subroutine call instruction.
code	Alias of CODE.
for	Assemble a “pushr” to start a FOR-NEXT loop.
FOR	Alias of FOR.

All short eP32 instruction assemblers are defined by “inst”. Their names are the same as mnemonics of respective machine instructions.

6.4 The eP32 Kernel

In the original eForth Model, a small group of FORTH commands were identified as kernel commands, low level commands, or primitive commands. These commands were coded in machine instructions of the host microprocessor. All other commands were written as lists of commands, and are called high level commands or compound commands. Compound commands are lists of primitive commands and other compound commands. This division of commands was very useful in porting eForth to many different microprocessors, because only primitive commands needed to be rewritten when moving eForth to a new microprocessor.

In eP32 eForth, we retained this division, and put primitive commands in the KERN32a.F file. However, we optimized commands in the eP32 so that the system executes at the highest speed and occupies the least memory space. All commands that can be are written in assembly. Much more optimization is achieved by a set of assembly macros, which assemble the most commonly used compound commands in machine instructions and pack these machine instructions as tightly as possible. The end results are that code size is significantly reduced and execution speed greatly increased.

Commands in this file also serve as programming examples for the optimal use of the eP32 CPU. It is worth your time to study them carefully, and use them as templates when you want to convert compound commands into assembly.

In the LatticeXP2-5E FPGA chip, there are 166K bits of Embedded Block Memory, EBM, and we use them to implement 4096 words of 32-bit RAM memory. The nicest feature of EBM is that it can be initialized from on-chip flash memory. In fact, this RAM memory can be used to host programs and data that otherwise would have to be implemented in ROM memory. This feature makes it possible to implement a complete FORTH system on a single FPGA chip, which has never been possible in other brands of FPGA.

Using EBM, the memory map of eP32 eForth is greatly simplified:

Address	Function
0x0	Reset and interrupt vectors
0x20	System variables
0x30	Text buffer
0x80	Terminal input buffer
0x100	Start of eForth dictionary
0x1FFF	End of RAM memory
0x80000000	Start of UART registers
0xE0000000	Start of GPIO registers

The data stack and return stack are in the eP32 core, and do not need RAM memory.

System variables are variables used by the eForth system to perform all its various functions. They are defined as assembly macro commands, with LDI machine instructions pointing to their respective addresses in the system variable area, starting at location $20. These assembly macro commands are tools used by the metacompiler to compile the optimized system variables referenced in the eP32 target system.

Command	Address	Function
HLD	20	Pointer to a buffer holding next digit for numeric conversion.
SPAN	21	Number of characters received by EXPECT.
>IN	22	Input buffer character pointer used by text interpreter.
#TIB	23	Number of characters in input buffer.
'TIB	24	Address of Terminal Input Buffer.
BASE	25	Number base for numeric conversion.
CONTEXT	26	Vocabulary array pointing to last name fields of vocabularies.
CP	27	Pointer to top of dictionary, the first available memory location.
LAST	28	Pointer to name field of last command in dictionary.
'EVAL	29	Execution vector switching between $INTERPRET and $COMPILE.
'ABORT	2A	Execution vector to handle error condition.
TEXT	30	Buffer to unpack text strings.
tmp	2B	Pointer to a scratch pad.
cpi	2C	Pointer to slots in assembler.
cpw	2D	Pointer to program word under construction.
etxbuf	80000001	Transmit data register.
etxbempty	80000001	Transmit status register.
erxbfull	80000002	Receiver status register.
erxbuf	80000003	Receiver data register.

```
HEX
cr .( system variables )
: HLD 20 ldi ;        \ scratch
: SPAN 21 ldi ;       \ #chars input by expect
: >IN  22 ldi ;       \ input buffer offset
: #TIB 23 ldi ;       \ #chars in the input buffer
: 'TIB 24 ldi ;       \ tib
: BASE 25 ldi ;      \ number base

cr
: CONTEXT 26 ldi ;    \ first search vocabulary
: CP 27 ldi ;        \ dictionary code pointer
: LAST 28 ldi ;      \ ptr to last name compiled
: 'EVAL 29 ldi ;      \ interpret/compile vector
: 'ABORT 2A ldi ;
: TEXT 30 ldi ;        \ unpack buffer
: tmp 2B ldi ;       \ ptr to converted # string
: cpi 2C ldi ;         \ assembler slot poiner
: cpw 2D ldi ;        \ pointer to word under construction

: etxbuf   80000001 ldi ;
: etxbempty 80000001 ldi ;
: erxbfull  80000002 ldi ;
: erxbuf   80000003 ldi ;

cr .( macro words ) cr
: DOLIT # ;
: EXIT ret ;
: EXECUTE ( a ) pushr ret anew ;
: ! ( n a -- ) tx stx ;
: @ ( a - n ) tx ldx ;
: R> ( - n ) popr ;
: R@ ( - n ) popr pushs pushr ;
: >R ( n ) pushr ;
: DUP ( n - n n ) pushs ;
: SWAP ( n1 n2 - n2 n1 )
   pushr tx popr xt ;
: DROP ( w w  -- )
   pops ;
: 2DROP ( w w  -- )
   pops pops ;
: + ( w w -- w ) add ;
: NOT ( w -- w ) com ;
: AND and ;
: XOR xor ;
: OVER over ;
: NEGATE ( n -- -n )
   com 1 ldi add ;
```

Assembly macro commands assemble one or more machine instructions into the target dictionary. One 32-bit program word can hold up to 5 short machine instructions. These assembly macro commands pack as many instructions in a program word as possible to make the most efficient use of memory and execution time. They allow the metacompiler to produce optimized code for the target system.

Macro	Function
DOLIT	Same as LIT. Assemble LDI; attach a value in next word.
EXIT	Single machine instruction.
EXECUTE	Push address in T to R and use RET to execute it.
!	Pop T to X and then store value in memory.
@	Pop T to X and then read value from memory.
R>	Single machine instruction.
R@	Pop R to T, duplicate T, and push T to R.
>R	Single machine instruction.
DUP	Single machine instruction.
SWAP	Use R and X to swap T and S.
DROP	Single machine instruction.
2DROP	Pop T twice.
+	Single machine instruction.
NOT	Single machine instruction.
AND	Single machine instruction.
XOR	Single machine instruction.
OVER	Single machine instruction.
NEGATE	Compliment T and add 1 to it.
1-	Add -1 to T.
1+	Add 1 to T.
BL	Return $20, ASCII code for space.
+!	Add n to contents of a. Pop a in T to X, fetch number, add n, and store back.
-	Subtract w3=w1-w2. Complement w2, add 1, and add w1.
OR	w3=w1 or w2. Complement w2, push it to R, complement w1, pop /w2, AND /w1, and complement results.
ROT	Rotate w1, w2, w3. Push w3, push w2, save w1 to X, pop w2, pop w3, and copy w1 back from X.
2DUP	Duplicate w1/w2 pair. Dup w2, push w2, push w2, dup w1, pop w1 to X, pop w2, push w1 from X, pop w2.
2!	Store double integer d in a. Pop address a to X, push dh, store dl, pop dh, and store dh.
2@	Fetch double integer from a. Pop address a to X, read dl, read dh.
COUNT	Retrieve n from a, and increment a. Pop address a to X, read n, push n, restore a+1 from X, pop n back.

```
: 1- ( a -- a )
   -1 ldi add ;
: 1+ ( a -- a )
   1 ldi add ;
: BL ( -- 32 )
   20 ldi ;
: +! ( n a -- )
   tx ldx add stx
   ;
: - ( w1 w2 -- w3 )
   com add 1 ldi add
   ;
: OR ( w1 w2 - w3 )
   com pushr com
   popr and com ;
: ROT ( w1 w2 w3 -- w2 w3 w1 )
   pushr pushr tx popr
   popr xt ;
: 2DUP ( w1 w2 -- w1 w2 w1 w2 )
   pushs pushr pushr
   pushs tx popr xt popr
   ;
: 2! ( d a -- )
   tx pushr stxp
   popr stx ;
: 2@ ( a -- d )
   tx ldxp ldx ;
: COUNT ( b -- b +n )
   tx ldxp pushr xt
   popr ;
cr .( kernel words ) cr
$100 org

code DOVAR popr ret
code 0< ( n - f )
   shl ifnc pushs pushs xor ret
   then
   -1 ldi ret
code UM+  ( n n - n carry )
   add pushs
   ifnc pushs pushs xor ret
   then
   1 ldi ret
code ?DUP ( w -- w w | 0 )
   pushs
   if pushs ret then
   ret
```

We are now actually compiling new commands into the target dictionary. First, assembly command ORG initializes the dictionary pointer, h, to memory location $100. The memory area below $100 is reserved for reset and interrupt vectors, system variables, text buffer, and the terminal input buffer.

The following are the first few code commands compiled into the eP32 target dictionary. They are defined using the CODE command, and when they are referenced later in the EP32q.F file, each of them will compile a subroutine call instruction pointing to their code field. The choice to define a CODE command as an assembly macro is rather arbitrary. However, if a command requires a branch instruction, it has to be coded as a CODE command, because macro commands cannot handle branch instructions gracefully. Assembly macro commands only do simple machine instruction placement.

Many compound commands defined in the original eForth model are now coded in assembly and moved to this kernel. We tried to do our best in giving you the smallest and fastest FORTH system. All commands that can be optimized are so optimized.

Command	Function
DOVAR	Execution code for variables. Return address of following program word. DOVAR is always followed by its value in the next program word, whose address happens to be in the R register. Pop return stack and this address is popped back onto the data stack.
0<	If n<0, return true flag; otherwise, return false flag. Negative flag is in bit T(31). Shift T left sends this bit into carry bit T(32), which is tested for branching by ifnc.
UM+	Add two integers on stack; return sum and carry. ADD adds two integers on data stack and carry bit is in T(32). “ifnc” tests this bit and pushs a 1 or 0 on stack accordingly.
?DUP	If w is not 0, duplicate it; otherwise, do nothing. w is duplicated and tested by “if”.
DNEGATE	Negate double integer d on stack. dh is first complemented and pushed onto the return stack. dl is complemented and incremented. If carry is set, dh is retrieved and incremented; otherwise, dh is retrieved but not incremented
ABS	Return absolute value of n. n is duplicated and tested for being negative by a left shift and “ifnc”. If negative, negate it; otherwise, leave it alone.
=	Return a true flag if the two numbers on data stack are equal; otherwise, return false flag. Use “xor” and “if” to test equality.
B>	Pack a byte at “b” into least significant 8 bits in “a”. Return b+1 and “a” to pack next byte.
>B	Unpack 4 bytes from “a” to byte array at “b”. Return a+1 and b+4 to unpack next word. Least significant byte in “a” is also returned, as it may be the count of a packed string.

```
cr
code DNEGATE ( d -- -d )
   com pushr com 1 ldi
   add pushs
   ifnc popr ret
   then
   popr 1 ldi add ret
code ABS ( n -- +n )
   pushs shl
   ifnc ret then
   NEGATE ret

cr
code = ( w w -- t )
   xor
   if pushs pushs xor ret then
   -1 ldi ret

cr ( pack b> and unpack >b strings )
code B> ( b a -- b+1 a )
   pushr tx ldxp pushr
   xt popr popr tx
   $FF ldi and
   ldx $FFFFFF00 ldi and xor
   rr8 stx xt ret
code >B ( a b -- a+1 b+4 count )
   pushr tx ldxp pushr
   xt popr popr ( a+1 n b ) tx
   pushs $FF ldi and stxp rr8
   pushs $FF ldi and stxp rr8
   pushs $FF ldi and stxp rr8
   pushs $FF ldi and stxp rr8
   pushr xt popr $FF ldi and
   ret
```

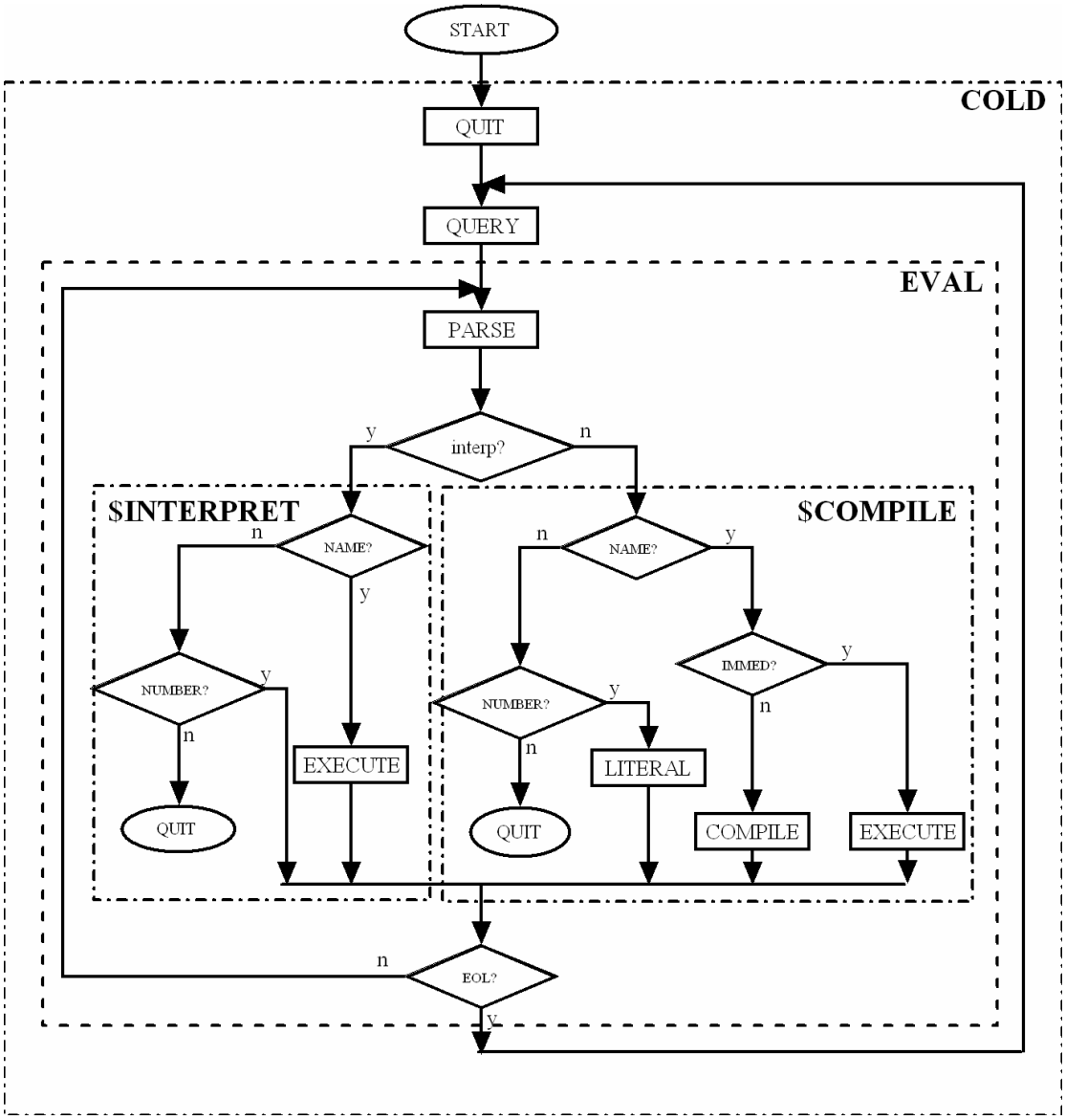

Figure 36. The eForth Operating System

6.5 eP32 Compound Commands

The EF32q.F. file contains compound commands to be compiled into the eP32 target image. These commands are defined with the "::" command and terminated by ";;" command. They are like the regular ":" and ";" commands in FORTH, but they compile new eP32 commands into the eP32 target image.

The ultimate goal of these commands is to implement an interactive operating system, or a text interpreter, which accepts a line of FORTH commands from a terminal, executes these commands in sequence, and waits for another line of commands. This FORTH system is best represented in the flowchart shown on the left page, in which all FORTH commands are enclosed in rectangles. As we go through source code in EP32q.F line by line, you will see how these commands are implemented, and will appreciate the overall design of this eP32 eForth system.

The text interpreter is also called the outer interpreter in FORTH. It is functionally equivalent to an operating system in a conventional microprocessor. It accepts commands similar to English words entered by a user, and carries out tasks specified by the commands. As an operating system, the text interpreter could be very complicated, because of all the things it has to do. However, because FORTH employs very simple syntax rules, and has very simple internal structures, the FORTH text interpreter is much simpler than conventional operating systems. It is simple enough that we can make a diagram of it as shown on the left page.

Let us summarize what a text interpreter does:

COLD	Power up routine
QUIT	Text interpreter
QUERY	Accept text input from a terminal
EVAL	Evaluate or interpret a line of text
PARSE	Parse out a string from input text
$INTERPRET	Interpret a string
$COMPILE	Compile a string
NAME$	Search dictionary for a command~~s~~
NUMBER?	Translate a text string into an integer
EXECUTE	Execute a command~~s~~
IMMED?	Is this command an immediate command?
LITERAL	Compile a integer literal
COMPILE	Compile a command token

FORTH allows us to build and integrate these functions gradually in modules. All modules finally fall into their places in the command QUIT, which is the text interpreter itself.

You might want to look up the code of QUIT first and see how the modules fit together. A good feeling for the big picture will help you in understanding lower modules. Nevertheless, we will doggedly follow the loading order in the source code, and hope that you will not get lost in the process.

```
 : :: code ;
: ;; ret ;

CRR .( Chararter IO ) CRR
:: ?KEY  erxbfull @ ;;
:: KEY   begin erxbfull @ until erxbuf @ ;;
:: EMIT  begin etxbempty  @ until etxbuf  ! ;;

CRR .( Common functions ) CRR
:: U< ( u u -- t ) 2DUP XOR 0< IF SWAP DROP 0< EXIT THEN - 0< ;;
::  < ( n n -- t ) 2DUP XOR 0< IF     DROP 0< EXIT THEN - 0< ;;
:: MAX ( n n -- n ) 2DUP     < IF SWAP THEN DROP ;;
:: MIN ( n n -- n ) 2DUP SWAP < IF SWAP THEN DROP ;;
:: WITHIN ( u ul uh -- t ) \ ul <= u < uh
   OVER - >R - R> U< ;;

CRR .( Divide ) CRR
CODE UM/MOD ( ud u -- ur uq )
   com 1 ldi add tx
   pushr xt pushr tx
   popr popr
   skip
CODE /MOD ( n n -- r q )
   com 1 ldi add pushr
   tx popr 0 ldi
   then
   div div div div
   div div div div
   div div div div
   div div div div
   div div div div
   div div div div
   div div div div
   div div div div
   div 1 ldi xor shr
   pushr pops popr xt
   ret
CODE MOD ( n n -- r )
   /MOD
   pops ret
CODE / ( n n -- q )
   /MOD
   pushr pops popr ret
:: M/MOD ( d n -- r q ) \ floored
  DUP 0<  DUP >R
  IF NEGATE >R DNEGATE R>
  THEN >R DUP 0< IF R@ + THEN R> UM/MOD R>
  IF SWAP NEGATE SWAP THEN ;;
```

Defining Compound Target Commands

::	Create a new compound target command. Because eForth uses the Subroutine Threading Model, compound commands and low level primitive commands are the same.
;;	Terminate a compound command. Assemble a RET machine instruction. All commands are called as subroutines, and RET will unnest a subroutine call, as well as a list of subroutine calls.

Character I/O

?KEY	Inspect register “erxbfull” and return a true flag if a character has been received. If no character was received, return a false flag.
KEY	Wait for a character, and return it after receiving it in “erxbuf”.
EMIT	Wait until transmit buffer is empty, by testing register “etxbempty”. Then send out a character to register “etxbuf”.

Common Functions

=	Return true if two integers are equal.
U<	Compare two unsigned integers. Return true if second integer is less than top integer. It is used to compare addresses.
<	Compare two signed integers. Return true if second integer is less than top integer.
MAX	Retain the larger of top two signed integers on stack.
MIN	Retain the lesser of top two signed integers on stack.
WITHIN	Check whether the third signed integer on stack is within range specified by top two signed integers. The range is inclusive of the lower limit and exclusive of the upper limit. If the third item is within range, a true flag is returned.

Divide

UM/MOD	Divide an unsigned double integer by an unsigned single integer. Return unsigned remainder and unsigned quotient. Unsigned double integer dividend is in the T:X register pair, and a negated 32-bit divisor is in the S register. Repeat “div” step 33 times. Remainder in the T register is shifted once too many, and it has to be shifted back one bit to the right.
/MOD	Divide a signed single integer by a signed integer. Return signed remainder and quotient.
MOD	Divide a signed single integer by a signed integer. Return signed remainder.
/	Divide a signed single integer by a signed integer. Return signed quotient.
M/MOD	Divide a signed double integer by a signed single integer. Return signed remainder and signed quotient.
M/	Divide a signed double integer by a signed single integer. Return signed quotient.

```
CRR .( Multiply ) CRR
CODE UM* ( u u -- ud )
   tx 0 ldi
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   pushr pops xt popr
   ret
:: * ( n n -- n ) UM* DROP ;;
:: M* ( n n -- d )
  2DUP XOR 0< >R  ABS SWAP ABS UM*  R> IF DNEGATE THEN ;;
:: */MOD ( n n n -- r q ) >R M* R> M/MOD ;;
:: */ ( n n n -- q ) */MOD SWAP DROP ;;

CRR .( Bits & Bytes ) CRR
:: >CHAR ( c -- c )
  $7F LIT AND DUP $7F LIT BL WITHIN
  IF DROP ( CHAR _ ) $5F LIT THEN ;;

CRR .( Memory access ) CRR
:: HERE ( -- a ) CP @ ;;
:: PAD ( -- a ) CP @ 50 LIT + ;;
:: TIB ( -- a ) 'TIB @ ;;
CRR
:: @EXECUTE ( a -- ) @ ?DUP IF EXECUTE THEN ;;
:: CMOVE ( b b u -- )
  FOR AFT >R DUP @ R@ ! 1+ R> 1+ THEN NEXT 2DROP ;;
:: FILL ( b u c -- )
  SWAP FOR SWAP AFT 2DUP ! 1+ THEN NEXT 2DROP ;;

:: PACK$ ( b u a -- a ) \ null fill
  pushs pushr
  2 ldi tmp tx stx
  tx pushs pushr rr8 stx
  xt popr
  FOR AFT ( b a )
    B>
    tmp tx ldx
    IF   ldx -1 ldi add stx
    ELSE 3 ldi stx
        1 ldi add
    THEN
  THEN NEXT
  BEGIN
     tx ldx $FFFFFF00 ldi and
     rr8 stx xt
     tmp tx ldx
  WHILE
     ldx -1 ldi add stx
  REPEAT
  pops pops popr
  ;;
```

Multiply

UM*	Multiply two unsigned integers and produce an unsigned double integer product. “mul” conditionally adds the integer in S to T if bit X(0) is set, and the T:X register pair is shift right by 1 bit. Two multiplicands are placed in the S and X registers. Repeat “mul” 32 times and a 64-bit product is produced in the T:X register pair.
*	Multiply two signed integers to produce a signed single integer product.
M*	Multiply two signed integers to produce a signed double integer product.
*/MOD	Multiply signed integers n1 and n2, and then divide the double integer product by n3. Scale n1 by n2/n3. Returns both remainder and quotient.
*/	Similar to */MOD except that it only returns quotient.

Bits and Bytes

>CHAR	Filter non-printable character to a harmless ‘underscore’ character, ASCII 95.

Memory Access

HERE	Returns address of WORD buffer 1 cell above command dictionary. Text interpreter parses out a string from Terminal Input Buffer and packs it here. In case this string is the name of a new command, it is already in the name field.
PAD	Returns address of a buffer pad 80 cells above command dictionary. It is a scratch pad area to store temporary text and data. It floats on top of the dictionary as new commands are added to the dictionary. The memory area below PAD is used for numeric conversion to build a number string backwards as least significant digits are extracted from an integer.
TIB	Return address of Terminal Input Buffer.
@EXECUTE	Jump to execution address stored in a memory location “a”.
CMOVE	Copy “u” cells of memory from array “b1” to array “b2”.
FILL	Fill “u” cells of memory array “b” with the same data, “c”.
PACK$	Copy “u” bytes in a byte array at “b” and pack them into a cell array at “a”. A packed string starts with a length byte in the lowest 8 bits of the first cell. PACK$ is designed to pack bytes into cells in a cell-addressable machine. The packed string is null-filled to a word boundary. Target address “a” is returned.

```
:: 4/
  shr shr ret
:: UNPACK$ ( a b -- b )
  DUP >R ( save b )
  >B $1F LIT AND 4/
  FOR AFT
    >B DROP
  THEN NEXT
  2DROP R>
  ;;
:: UNPACK ( a b -- b )
  DUP >R ( save b )
  >B $FF LIT AND 4/
  FOR AFT
    >B DROP
  THEN NEXT
  2DROP R>
  ;;

CRR .( Numeric Output ) CRR \ single precision
:: DIGIT ( u -- c )
  9 LIT OVER < 7 LIT AND +
  ( CHAR 0 ) 30 LIT + ;;
:: EXTRACT ( n base -- n c )
  0 LIT SWAP UM/MOD SWAP DIGIT ;;
:: <# ( -- ) PAD HLD ! ;;
:: HOLD ( c -- ) HLD @ 1- DUP HLD ! ! ;;
:: # ( u -- u ) BASE @ EXTRACT HOLD ;;
:: #S ( u -- 0 ) BEGIN # DUP WHILE REPEAT ;;
CRR
:: SIGN ( n -- ) 0< IF ( CHAR - ) 2D LIT HOLD THEN ;;
:: #> ( w -- b u ) DROP HLD @ PAD OVER - ;;
:: str ( n -- b u ) DUP >R ABS <# #S R> SIGN #> ;;
:: HEX ( -- ) 10 LIT BASE ! ;;
:: DECIMAL ( -- ) 0A LIT BASE ! ;;

CRR .( Numeric Input ) CRR \ single precision
:: DIGIT? ( c base -- u t )
  >R ( CHAR 0 ) 30 LIT - 9 LIT OVER <
  IF 7 LIT - DUP 0A LIT  < OR THEN DUP R> U< ;;
:: NUMBER? ( a -- n T | a F )
  BASE @ >R  0 LIT OVER COUNT ( a 0 b n)
  OVER @ ( CHAR $ ) 24 LIT =
  IF HEX SWAP 1+ SWAP 1- THEN ( a 0 b' n')
  OVER @ ( CHAR - ) 2D LIT = >R ( a 0 b n)
  SWAP R@ - SWAP R@ + ( a 0 b" n") ?DUP
  IF 1- ( a 0 b n)
    FOR DUP >R @ BASE @ DIGIT?
      WHILE SWAP BASE @ * +  R> 1+
    NEXT DROP R@ ( b ?sign) IF NEGATE THEN SWAP
      ELSE R> R> ( b index) 2DROP ( digit number) 2DROP 0 LIT
      THEN DUP
  THEN R> ( n ?sign) 2DROP R> BASE ! ;;
```

4/	Divide top of stack by 4.
UNPACK$	Unpacks a packed string at “a” to a byte array at “b”. The first byte in the packed string is a length byte. Unpack only up to 31 bytes. Use >B to do unpacking.
UNPACK	Identical to UNPACK$, except it unpacks strings up to 255 bytes.

Numeric Output

FORTH is interesting in its special capabilities in handling numbers across a man-machine interface. It recognizes that machines and humans prefer very different representations of numbers. Machines prefer binary representation, but humans prefer decimal Arabic representation. However, depending on circumstances, a human may want numbers to be represented in other radices, like hexadecimal, octal, and sometimes binary.

FORTH solves this problem of internal (machine) versus external (human) number representations by insisting that all numbers are represented in binary form in CPU and memory. Only when numbers are imported or exported for human consumption are they converted to external ASCII representation. The radix of the external representation is stored in system variable BASE. The user can select any reasonable radix in BASE, up to 72, limited by available printable characters in the ASCII character set.

DIGIT	Convert integer “u” to a digit “c”.
EXTRACT	Extract least significant digit “c” from a number “n”. “n” is divided by radix “base”.
HOLD	Insert an ASCII character “c” in numeric output string.
"#"	Extract one digit from integer “u”, according to radix in BASE, and add it to output string.
"#S"	Extract all digits to output string until “u” is 0.
SIGN	Insert a “-” sign in numeric output string if “n” is negative.
#>	Terminate numeric conversion and return address and length of output string.
str	Convert signed integer “n” to a numeric output string.
HEX	Set numeric conversion radix to 16 for hexadecimal conversions.
DECIMAL	Set numeric conversion radix to 10 for decimal conversions.

Numeric Output

DIGIT?	Convert a digit “c” to its numeric value “u” according to current radix “b”. If conversion is successful, push a true flag above “u”. If not successful, return “c” and a false flag.
NUMBER?	Convert a count string of digits at location “a” to an integer. If first character is a $, convert in hexadecimal; otherwise, convert using radix in BASE. If first character is a “-”, negate integer. If an illegal character is encountered, address of string and a false flag are returned. Successful conversion returns integer value and a true flag.

```
CRR .( Basic I/O ) CRR
:: SPACE ( -- ) BL EMIT ;;
:: CHARS ( +n c -- )
  SWAP 0 LIT MAX
  FOR AFT DUP EMIT THEN NEXT DROP ;;
:: SPACES ( +n -- ) BL CHARS ;;
:: TYPE ( b u -- )
  FOR AFT DUP @ >CHAR EMIT 1+
  THEN NEXT DROP ;;
:: CR ( -- ) ( =Cr )
  0A LIT 0D LIT EMIT EMIT ;;
:: do$ ( -- a )
  R> R@ TEXT UNPACK
  R@ R> @ $FF LIT AND 4/ 1+ +
  >R SWAP >R ;;

CRR
:: $"| ( -- a ) do$ ;;
:: ."| ( -- ) do$ COUNT TYPE ;;
::  .R ( n +n -- )
  >R str     R> OVER - SPACES TYPE ;;
:: U.R ( u +n -- )
  >R <# #S #> R> OVER - SPACES TYPE ;;
:: U. ( u -- ) <# #S #> SPACE TYPE ;;
::  . ( n -- )
  BASE @ 0A LIT  XOR
  IF U. EXIT THEN str SPACE TYPE ;;
:: ? ( a -- ) @ . ;;

CRR .( Parsing ) CRR
:: (parse) ( b u c -- b u delta ; <string> )
  tmp ! OVER >R DUP \ b u u
  IF 1- tmp @ BL =
    IF            \ b u' \ 'skip'
      FOR BL OVER @ - 0< NOT
        WHILE 1+
      NEXT ( b) R> DROP 0 LIT DUP EXIT \ all delim
        THEN  R>
    THEN OVER SWAP  \ b' b' u' \ 'scan'
    FOR tmp @ OVER @ -  tmp @ BL =
      IF 0< THEN WHILE 1+
    NEXT DUP >R
      ELSE R> DROP DUP 1+ >R
      THEN OVER -  R>  R> - EXIT
  THEN ( b u) OVER R> - ;;
:: PARSE ( c -- b u ; <string> )
  >R  TIB >IN @ +
  #TIB @ >IN @ -
  R> (parse) >IN +! ;;
:: TOKEN ( -- a ;; <string> )
  BL PARSE 1F LIT MIN 2DUP
  DUP TEXT ! TEXT 1+ SWAP CMOVE
  HERE 1+ PACK$ ;;
:: WORD ( c -- a ; <string> )
  PARSE HERE 1+ PACK$ ;;
```

Basic I/O

SPACE	Output a blank space character.
SPACES	Output “n” blank space characters.
CHARS	Output a string of “n” characters “c”.
CR	Output a carriage-return and a line-feed.
TYPE	Output “n” characters from a string in memory “b”.
do$	Unpack a packed string literal, pointed to by address on return stack. The string is unpacked to TEXT buffer “a”. The return address on return stack is incremented to skip over the string literal.

String literals are data structures compiled in compound commands, in-line with other commands. A string literal must start with a string command, which knows how to handle the following packed string at run time.

$"|	Alias of "do$. Unpack following packed string in this string literal and return address of unpacked string.
."|	Unpack following packed string in this string literal and output string characters.
.R	Output a signed integer “n” right-justified in a field of “+n” characters.
U.R	Output an unsigned integer “n” right-justified in a field of “+n” characters.
U.	Output an unsigned integer “u” in free format, followed by a space.
.	Output a signed integer “n” in free format, followed by a space.
?	Output a signed integer stored in memory “a”, in free format followed by a space.

Parsing

FORTH source code consists of commands, which are ASCII strings separated by spaces and other white space characters like tabs, carriage returns, and line feeds. The text interpreter scans text in the Terminal Input Buffer, TIB, isolates strings and interprets them in sequence. After a string is parsed out of the input stream, the text interpreter “interprets” it—executes it if it is a command, compiles it if the text interpreter is in compiling mode—and converts it to a number if the string is not a valid command.

(parse)	Parse out the first string delimited by character “c” from input buffer at b1, length u1. Return address b2 and length u2 of the string just parsed out, and the difference “n” between b1 and b2.
PARSE	Parse a string delimited by character “c” in TIB, from character pointed to by >IN. It returns address “b” and the length of parsed string “u”.
TOKEN	Parse out next text string delimited by a space character in TIB. The text string is assumed to be the name of a command, and its length is limited to 31 characters. This string is packed into the WORD buffer one cell above the dictionary; i.e., HERE+1.
WORD	Parse out next text string delimited by character “c” in TIB. This string is packed into the WORD buffer one word above the command dictionary; i.e., HERE+1. Length of string is limited to 255 characters.

```
CRR .( Dictionary Search ) CRR
:: NAME> ( a -- xt )
  DUP @ $1F LIT AND
  4/ + 1+ ;;
:: SAME? ( a1 a2 u -- a1 a2 f \ -0+ )
  $1F LIT AND 4/
  FOR AFT OVER R@ + @
    OVER R@ + @ - ?DUP
    IF R> DROP EXIT THEN
  THEN NEXT
  0 LIT ;;
:: find ( a va -- xt na | a F )
  SWAP        \ va a
  DUP @ tmp !  \ va a  \ get cell count
  DUP @ >R     \ va a  \ count
  1+ SWAP      \ a' va
  BEGIN @ DUP  \ a' na na
    IF DUP @ $FFFFFF3F LIT AND
      R@ XOR \ ignore lexicon bits
      IF 1+ -1 LIT
      ELSE 1+ tmp @ SAME?
      THEN
    ELSE R> DROP SWAP 1- SWAP EXIT \ a F
    THEN
  WHILE 1- 1-  \ a' la
  REPEAT R> DROP SWAP DROP
  1- DUP NAME> SWAP ;;
:: NAME? ( a -- xt na | a F )
  CONTEXT find ;;

CRR .( Terminal ) CRR
:: ^H ( bot eot cur -- bot eot cur ) \ backspace
  >R OVER R> SWAP OVER XOR
  IF ( =BkSp ) 8 LIT EMIT
     1-        BL EMIT \ distructive
     ( =BkSp ) 8 LIT EMIT \ backspace
  THEN ;;
:: TAP ( bot eot cur c -- bot eot cur )
  DUP EMIT OVER ! 1+ ;;
:: kTAP ( bot eot cur c -- bot eot cur )
  DUP ( =Cr ) 0D LIT XOR
  IF ( =BkSp ) 8 LIT XOR
    IF BL TAP ELSE ^H THEN
    EXIT
  THEN DROP SWAP DROP DUP ;;

CRR
:: accept ( b u -- b u )
  OVER + OVER
  BEGIN 2DUP XOR
  WHILE  KEY  DUP BL -  5F LIT U<
    IF TAP ELSE kTAP THEN
  REPEAT DROP  OVER - ;;
:: EXPECT ( b u -- ) accept SPAN ! DROP ;;
:: QUERY ( -- )
  TIB 50 LIT accept #TIB !
  DROP 0 LIT >IN ! ;;
```

Dictionary Search

In this FORTH system, records of commands are linked into a command dictionary. A record contains three fields: a link field holding the name field address of the previous record, a name field holding the name of this command as a packed string, and a code field holding the executable code of this command. The command dictionary is a linear list linked through link fields and the name fields of all records.

NAME>	Return code field address “xt” from name field address “a” of a command.
SAME?	Compare two packed strings at “a1” and “a2” for “u” cells. If string1>string2, returns a positive integer. If string1<string2, return a negative integer. If strings are identical, return a 0.
find	Look up a packed string at “a” in command dictionary. Search starts at “va”. If a command is found, return code field address “xt” and name field address “na”. If the string is not found, return address “a” and a false flag.
NAME?	Search dictionary from CONTEXT for a name at “a”. Return code field address and name field address if a command is found. Otherwise, return address “a” and a false flag.

Terminal

The text interpreter interprets source text received from an input device and stored in the Terminal Input Buffer. To process characters in the Terminal Input Buffer, we need special commands to deal with the special conditions of backspace character and carriage return:

^H	Process back-space. Erase last character and decrement “cur”. If “cur”=”bot”, do nothing because you cannot backup beyond beginning of input buffer.
TAP	Output character “c” to terminal, store “c” in “cur”, and increment “cur”, which points to the current character. “bot” and “eot” are the beginning and end of the input buffer.
kTAP	Processes character “c”. “bot” is the beginning of the input buffer, and “eot” is the end. “cur” points to the current character in the input buffer. “c” is normally stored at “cur”, which is incremented by 1. If “c” is a carriage-return, echo a space and make “eot”=”cur”. If “c” is a back-space, erase the last character and decrement “cur”.
accept	Accept “u” characters into buffer at “b”, or until a carriage return. The value of “u” returned is the actual count of characters received.
EXPECT	Accept “u” characters into buffer at “b”, or until a carriage return. The count of characters received is in SPAN.
QUERY	Accept up to 80 characters from the input device to the Terminal Input Buffer. This also prepares the Terminal Input Buffer for parsing by setting #TIB to characters received and clearing >IN, pointing to the beginning of the Terminal Input Buffer.

```
CRR .( Error handling ) CRR
:: ABORT ( -- ) 'ABORT @EXECUTE ;;
:: abort" ( f -- )
  IF do$ COUNT TYPE ABORT THEN do$ DROP ;;

CRR .( Interpret ) CRR
:: ERROR ( a -- )
  DROP SPACE TEXT COUNT TYPE
  $3F LIT EMIT CR ABORT
:: $INTERPRET ( a -- )
  NAME?  ?DUP
  IF @ $40 LIT AND
    abort" $LIT compile only" EXECUTE EXIT
  THEN DROP TEXT NUMBER?
  IF EXIT THEN ERROR
:: [ ( -- )
  forth_' $INTERPRET  >body forth_@ LIT 'EVAL !
  ;; IMMEDIATE
:: .OK ( -- )
  forth_' $INTERPRET >body forth_@ LIT 'EVAL @ =
  IF ."| $LIT  OK" CR
  THEN ;;
:: EVAL ( -- )
  BEGIN TOKEN DUP @
  WHILE 'EVAL @EXECUTE \ ?STACK
  REPEAT DROP .OK ;;

CRR .( Shell ) CRR
:: QUIT ( -- )
  ( =TIB) $80 LIT 'TIB !
   [ BEGIN QUERY EVAL AGAIN

CRR .( Compiler Primitives ) CRR
:: ' ( -- xt )
  TOKEN NAME? IF EXIT THEN
  ERROR
:: ALLOT ( n -- ) CP +! ;;
:: , ( w -- ) HERE DUP 1+ CP ! ! ;;
:: [COMPILE] ( -- ; <string> )
  '  4000000 LIT + , ;; IMMEDIATE

CRR
:: COMPILE ( -- ) R> DUP @ , 1+ >R ;;
:: LITERAL $A79E79E LIT , ,
  ;; IMMEDIATE
:: $," ( -- ) ( CHAR " )
  22 LIT WORD
  DUP @ $FF LIT AND
  4/ + 1+ CP ! anew ;;
:: (CALL) ( a -- call ) FFFFFF LIT AND 4000000 LIT OR ;;
```

Interpreter

ABORT	Execute the command whose address is in the system variable 'ABORT. This address normally points to QUIT.
abort"	When the top item on stack is non-zero, output the following packed string and execute ABORT; otherwise, skip over error message. It is compiled before a packed error message.
ERROR	Display error message in TEXT buffer and execute ABORT.
[	Activate interpreting mode by storing $INTERPRET into variable 'EVAL, which is executed in EVAL.
.OK	Prints the OK prompt. OK is printed only when the text interpreter is in interpreting mode. While compiling, the OK prompt is suppressed.
EVAL	Interpreter loop, which parses strings from the Terminal Input Buffer, and the command in 'EVAL to process a string, either executing it with $INTERPRET or compiling it with $COMPILE.
$INTERPRET	Processes a string at “a”. If it is a valid command, execute it; otherwise, convert it to a number. Failing that, execute ERROR and return to QUIT.

Compiler Primitives

‘	Search dictionary for following name, and return its code field address if a command is found; otherwise, print a warning message with “?”.
ALLOT	Allocate “n” cells of memory on top of dictionary.
,	Compile an integer “w” to dictionary, and add the new item to the growing command list of the current command under construction. This is the primitive compiler.
[COMPILE]	Compile the code field address of the next command. It compiles an immediate command, even if it would otherwise be executed.
COMPILE	Compile the code field address of the next command. It forces compilation of a command at run time.
LITERAL	Compile an integer literal. It first compiles doLIT, followed by an integer vale from the stack. When doLIT is executed, it extracts the integer in the next program word and pushes it on the stack.
$,	Compile a packed string. String text is taken from the input stream and terminated by a double quote. A token (such as . "| or $"|) must be compiled before the string to form a sting literal.
(CALL)	Compile or assemble a subroutine CALL instruction with the code field address on the stack. Compound commands are compiled as lists of subroutine calls.

```
CRR .( Name Compiler ) CRR
:: ?UNIQUE ( a -- a )
  DUP NAME?
  IF TEXT COUNT TYPE ."| $LIT reDef "
  THEN DROP ;;
:: $,n ( a -- )
  DUP @
  IF ?UNIQUE
    ( na) DUP NAME> CP !
    ( na) DUP LAST ! \ for OVERT
    ( na) 1-
    ( la) CONTEXT @ SWAP ! EXIT
  THEN ERROR

CRR .( FORTH Compiler ) CRR

:: $COMPILE ( a -- )
  NAME? ?DUP
  IF @ $80 LIT AND
    IF EXECUTE
    ELSE (CALL) , anew
    THEN EXIT
  THEN DROP TEXT NUMBER?
  IF LITERAL anew EXIT
  THEN ERROR
:: OVERT ( -- ) LAST @ CONTEXT ! ;;
:: ; ( -- )
  $179E79E LIT , [ OVERT ;; IMMEDIATE
:: ] ( -- )
  forth_' $COMPILE >body forth_@ LIT 'EVAL ! ;;
:: : ( -- ; <string> )
  TOKEN $,n ] ;;

CRR .( Tools ) CRR
:: dm+ ( b u -- b )
  OVER 6 LIT U.R SPACE
  FOR AFT DUP @ 9 LIT U.R 1+
  THEN NEXT ;;
:: DUMP ( b u -- )
  BASE @ >R HEX  8 LIT /
  FOR AFT CR 8 LIT dm+
  THEN NEXT DROP R> BASE ! ;;

CRR
:: >NAME ( xt -- na | F )
  CONTEXT
  BEGIN @ DUP
  WHILE 2DUP NAME> XOR
    IF 1-
    ELSE SWAP DROP EXIT
    THEN
  REPEAT SWAP DROP ;;
:: .ID ( a -- )
  TEXT UNPACK$
  COUNT $01F LIT AND TYPE SPACE ;;
```

Name Compiler

?UNIQUE	Display a warning message to show that the name of a new command is the same as a command already in the dictionary.
$,n	Build a new header in the dictionary using the name string already packed in the WORD buffer. Fill in the link field with the address in LAST. The top of the dictionary is now the code field of a new command, ready to accept commands and tokens.
$COMPILE	Process a string at “a”, and compile a new token, a call instruction, in the dictionary. This dictionary pointer in CP is incremented, and is ready to compile the next token.
OVERT	Link a new command to the dictionary and make it available for a dictionary search. OVERT changes CONTEXT to point to the name field of this new command, and extends the dictionary chain to include a new command.
;	Terminate a compound command. Compile a RET instruction to terminate a token list. Link this command to the dictionary, and change the text interpreter to interpreting mode.
]	Activate compiling mode by writing the address of $COMPILE into system variable 'EVAL.
:	Create a new compound command. Take the next input string to build a new header. Now, its code field is on top of the command dictionary, and is ready to accept new tokens.

Tools

dm+	Display 8 words from address “b”. Return new address b+8 for the next dm+.
DUMP	Display “u” words from address “b”, with 8 words on a line. A line begins with an address, followed by 8 words in hex.

Decompiler Tools

Since name fields are linked into a list in the command dictionary, it is fairly easy to locate a command by searching its name in the command dictionary. However, finding the name of a command from its code field address is more difficult, because the name field has variable length, and we cannot scan the name field backwards very easily.

>NAME	Return a code field address, “xt”, of a command from its name field address, “na”. If “xt” is not a valid code field address, return 0. It follows the linked list of the command dictionary, and from every name field address we can get a corresponding code field address. If this address is not the same as “xt”, we go to the name field of the next command. If “xt” is a valid code field address, we surely will find it. If the entire dictionary is searched and “xt” is not found, it is not a valid code field address.
.ID	Display the name of a command, given its name field address “a”. It replaces non-printable characters in a name by underscores.

```
CRR
:: SEE ( -- ; <string> )
  ' CR
  BEGIN
    20 LIT FOR
      DUP @ DUP 3F000000 LIT AND
      4000000 LIT XOR
      IF U. SPACE
      ELSE FFFFFF LIT AND >NAME
        ?DUP IF .ID THEN
      THEN 1+
    NEXT KEY 0D LIT =  \ can't use ESC on terminal
  UNTIL DROP ;;
:: WORDS ( -- )
  CR CONTEXT
  BEGIN @ ?DUP
  WHILE DUP SPACE .ID 1-
  REPEAT ;;
CODE .S ( dump all 33 stack items )
  PAD tx stxp
  stxp stxp stxp stxp
  stxp stxp stxp stxp
  stxp stxp stxp stxp
  stxp stxp stxp stxp
  stxp stxp stxp stxp
  stxp stxp stxp stxp
  stxp stxp stxp stxp
  stxp stxp stxp stxp
  PAD $21 LIT
  FOR DUP ? 1+ NEXT
  DROP PAD @ CR ;;

CRR .( file download and upload ) CRR
:: READ PAD
  BEGIN KEY DUP 1A LIT XOR
  WHILE OVER ! 1+
  REPEAT DROP
  PAD - SPAN ! ;;
:: OK 'TIB @ >R #TIB @ >R >IN @ >R
  PAD 'TIB ! SPAN @ #TIB ! 0 LIT >IN !
  EVAL R> >IN ! R> #TIB ! R> 'TIB ! ;;
:: SEND ( b u -- )
  CR
  FOR AFT DUP @ <# # # # # # # # # #> TYPE 1+
    DUP 7 LIT AND IF SPACE ELSE CR THEN
  THEN NEXT
  DROP ;;
:: FORGET ( -- )
  TOKEN NAME? ?DUP
  IF 1- DUP CP !
     @ DUP CONTEXT ! LAST !
     DROP EXIT
  THEN ERROR
```

SEE	Search the next word in the input stream for a command, and decompile the first 32 program words in its code field. Display an error message if the next word is not a valid command. It scans the code field and looks for CALL instructions. If it finds a CALL instruction, use the address in the address field to find this command in the command dictionary, and display its name. If a word in the code field is not a CALL instruction, just display its value.
WORDS	Display all names in the command dictionary. The display order of commands is reversed from compiling order. The last defined command is displayed first.
.S	Display the contents of the data stack on screen in free format. The bottom of the stack is shown on the right. The topitem is shown on the left. The eP32 has a 33-level hardware data stack in the CPU, and it wraps around like a circular buffer. .S displays all 32 stack levels and the T register.

File Download and Upload

If the eForth system is connected to the serial port of a computer, the computer can emulate a terminal to communicate with eForth. Most terminal emulation programs can send large text files to the serial port. The user can now compose and edit large applications as text files on the computer. The text file can then be downloaded to eForth for interpreting or compiling.

PAD is a free memory area 80 words above the top of the command dictionary. It can be used to store temporary data, and is an ideal place to download a text file.

READ	Accept characters from terminal and store them in PAD buffer. A Ctrl-Z character terminates the READ command. After a file is downloaded, the length of the file is stored in variable SPAN.
OK	Interpret text downloaded in PAD buffer. In QUIT, EVAL interprets text in the Terminal Input Buffer. EVAL uses three system variables to manage the Terminal Input Buffer: 'TIB points to the beginning of the text buffer, #TIB contains the length of the text, and >IN points to a character in the text buffer currently being interpreted. OK saves these three variables, replaces them by PAD, SPAN, and a 0, and then calls EVAL to interpret the text in the PAD buffer. After the text is interpreted successfully, 'TIB, #TIB and >IN are restored and the text interpreter is restored to its normal state.
SEND	Upload contents of a memory area, “n” words starting at address “b”, to the terminal. Each word is sent as 8 hex digits, followed by a space. A carriage return-linefeed pair is sent every 8 words.
FORGET	Search the next word in the input stream for a command. If it is a valid command, delete it and all subsequent command records from the dictionary.

```
CRR .( Hardware reset ) CRR
::   DIAGNOSE    ( - )
     $65 LIT
\ 'F'  prove UM+ 0<        \ carry, TRUE, FALSE
     0 LIT 0< -2 LIT 0<     \ 0 FFFF
     UM+ DROP             \ FFFF ( -1)
     3 LIT UM+ UM+ DROP     \ 3
     $43 LIT UM+ DROP      \ 'F'
\ 'o' logic: XOR AND OR
     $4F LIT $6F LIT XOR    \ 20h
     $F0 LIT AND
     $4F LIT OR
\ 'r' stack: DUP OVER SWAP DROP
      8 LIT 6 LIT SWAP
      OVER XOR 3 LIT AND AND
      $70 LIT UM+ DROP      \ 'r'
\ 't'-- prove BRANCH ?BRANCH
      0 LIT IF $3F LIT THEN
      -1 LIT IF $74 LIT ELSE $21 LIT THEN
\ 'h' -- @ ! test memeory address
      $68 LIT $40 LIT !
      $40 LIT @
\ 'M' -- prove >R R> R@
      $4D LIT >R R@ R> AND
\ 'l'  -- prove 'next' can run
      61 LIT $A LIT FOR 1 LIT UM+ DROP NEXT
\ 'S' -- prove ldp, stp
      $50 LIT $3 LIT
      $30 LIT tx stxp stxp
      $30 LIT tx ldxp ldxp
      xor
\ 'emi' -- prove mul, dupy, popy
      $656D LIT $1000000 LIT UM*
      SWAP rr8 rr8 rr8
\ ' C' -- prove div
      $2043 LIT 0 LIT $100 LIT UM/MOD
\     ;;

CRR
:: COLD ( -- )
   DIAGNOSE
   CR ."| $LIT eP32q v"
   DECIMAL
   CC LIT <# # # ( CHAR . ) 2E LIT HOLD # #> TYPE
   CR QUIT
```

Hardware Reset

When eP32 is powered up, or when it is reset, it executes COLD to start the eForth system running. The first thing COLD does is call a diagnostic routine, DIAGNOSE, to run a series of tests, verifying that the eP32 core is working properly. It is superfluous once the eP32 is fully debugged. However, in implementing the eP32 on a new FPGA or on a custom chip, DIAGNOSE is extremely helpful in hardware simulation and in hardware verification. In about 1000 cycles, you can observe most instructions executed, and verify that they execute correctly.

DIAGNOSE tests the following machine and primitive commands in the eP32:
LIT
0<
BZ
UM+
DROP
XOR
AND
OR
DUP
OVER
SWAP
BRA
@
!
>R
R@
R>
NEXT
TX
STXP
LDXP
RR8
UM*
UM/MOD

Cold Boot

COLD initializes the eP32 to start eForth. The eP32 is a real FORTH microprocessor, and the hardware initializes itself. COLD does not have much to do. It first executes DIAGNOSE to run a few tests on eP32 machine instructions, displays a sign-on message, and then jumps to QUIT. COLD is the first compound command executed after power up or after chip reset. Its address is placed in memory location 0, which is the hardware reset vector.

```
CRR .( Structures ) CRR
:: BEGIN ( -- a ) anew HERE ;; IMMEDIATE
:: THEN ( A -- ) BEGIN SWAP +! ;; IMMEDIATE
:: FOR ( -- a ) 1C79E79E LIT , BEGIN ;; IMMEDIATE
CRR
:: NEXT ( a -- )  5000000 LIT OR , anew ;; IMMEDIATE
:: UNTIL ( a -- ) 2000000 LIT OR , anew ;; IMMEDIATE
:: AGAIN ( a -- ) 0000000 LIT OR , anew ;; IMMEDIATE
:: IF ( -- A )  BEGIN 2000000 LIT , ;; IMMEDIATE
CRR
:: AHEAD ( -- A ) BEGIN 0000000 LIT , ;; IMMEDIATE
:: REPEAT ( A a -- ) AGAIN THEN ;; IMMEDIATE
:: AFT ( a -- a A ) DROP AHEAD BEGIN SWAP ;; IMMEDIATE
:: ELSE ( A -- A )  AHEAD SWAP THEN ;; IMMEDIATE
:: WHEN ( a A -- a A a ) IF OVER ;; IMMEDIATE
:: WHILE ( a -- A a )   IF SWAP ;; IMMEDIATE

CRR
:: ABORT" ( -- ; <string> )
   forth_' abort" >body forth_@ LIT (CALL) HERE !
   $," ;; IMMEDIATE
:: $" ( -- ; <string> )
   forth_' $"| >body forth_@ LIT (CALL) HERE !
   $," ;; IMMEDIATE
:: ." ( -- ; <string> )
   forth_' ."| >body forth_@ LIT (CALL) HERE !
   $," ;; IMMEDIATE

CRR
': doVAR popr ret
:: CODE ( -- ; <string> ) TOKEN $,n OVERT align ;;
:: CREATE ( -- ; <string> ) CODE
    forth_' doVAR >body forth_@ LIT (CALL) , ;;
:: VARIABLE ( -- ; <string> ) CREATE 0 LIT , ;;
:: CONSTANT CODE $A040000  LIT , , ;;
:: DOES ( -- ) R> (CALL) LAST @ NAME> ! ;;
```

Structures

BEGIN	Begin a loop structure. Leave address “a” of the current program word on the stack.
THEN	Resolve address field in a transfer instruction at “a”.
FOR	Assemble a PUSH instruction and leave the address of the next word “a” on the stack.
NEXT	Assemble a NEXT instruction using target address “a”.
UNTIL	Assemble a BZ instruction using target address “a”.
AGAIN	Assemble a BRA instruction using target address “a”.
IF	Assemble a BZ instruction whose address, “a”, is left on the stack.
AHEAD	Assemble a BRA instruction whose address, “a”, is left on the stack.
REPEAT	Assemble a BRA instruction using target address “a”. Use the address of the next program word to resolve the address field of the branch instruction at “a”..
AFT	Assemble a BZ instruction and leave its address as “a”,. Replace the address “a” left by FOR with the address of the next program word.
ELSE	Assemble a BRA instruction, and use the address of the next program word to resolve the address field of the BZ instruction in “a”.. Replace “a”with the address of its BRA instruction.
WHILE	Assemble a BZ instruction and leave its address, “a”, on the stack. Address “a” is swapped to the top of the data stack.

String Commands

ABORT"	Compile an error message. This error message is displayed when the top of the stack is non-zero.
."	Compile a string literal, which will be displayed at run time.
$"	Compile a string literal. When it is executed, only the address of the string is left on the data stack for the next commands to access this string.

Defining Commands

Defining commands are molds to create many commands that share the same run time execution behavior.

CODE	Create a new primitive command that is intended to contain machine instructions.
:	Create a new compound command to compile a tokens list. The text interpreter is switched to compiling mode, which handles integer literals and control structures more gracefully.
CREATE	Create a new data array without allocating memory.
VARIABLE	Create a new variable, initialized to 0.
CONSTANT	Create an integer constant.
DOES	Define the run time execution routine for a new class of commands. This execution routine follows the DOES command. It is similar to the DOES> command that we used in the assembler.

```
CRR
(makehead) .( ( -- ) 29 LIT PARSE TYPE ;; IMMEDIATE
(makehead) \ ( -- ) #TIB @ >IN ! ;; IMMEDIATE
(makehead) ( 29 LIT PARSE 2DROP ;; IMMEDIATE
(makehead) IMMEDIATE $80 LIT LAST +! ;;

CRR
(makehead) EXIT popr pops ret
(makehead) EXECUTE pushr ret
(makehead) ! tx stx ret
(makehead) @ tx ldx ret
(makehead) R> popr tx popr xt pushr ret
(makehead) R@ popr tx popr pushs pushr xt pushr ret
(makehead) >R popr tx pushr xt pushr ret
(makehead) SWAP
  pushr tx popr xt ret
(makehead) OVER
  pushr pushs tx popr
  xt ret
(makehead) 2DROP
  pops pops ret

(makehead) +  add ret
(makehead) NOT com ret
(makehead)  NEGATE
  com 1 ldi add ret
(makehead) 1-
  -1 ldi add ret
(makehead) 1+
  1 ldi add ret
```

Makehead Commands

(makeHead) compiles only a header in the target dictionary and such commands are invisible to the metacompiler. In contrast, the "::" command compiles a header in the target dictionary and a header in the metacompiler, and the command thus defined will compile itself to the target dictionary when subsequently invoked. After (makehead) commands are defined in the target dictionary, they can still be used in the metacompiler as usual.

.(	Display the following string, delimited by).
\	Start a comment. Ignore all characters until end of line.
(	Start a comment. Ignore the following string, delimited by).
IMMEDIATE	Set the immediate bit in the name field of the last defined command. Such a command will be executed, not compiled, in compiling mode.

Redefine Macro Commands

A set of macro commands were defined in eP32 assembler to produce optimized code in the eForth system. These commands are also needed in the target system. Here they are re-defined as primitive commands for the eP32 target system. In the eForth target, they will be compiled as a subroutine call without optimization. To produce optimized code for the target, we need an optimizing assembler for the target. It was so implemented in one of our earlier eP32 systems, and was fairly complicated. We decide to leave it out for this XP2 FPGA implementation.

Command	Function
EXIT	Return from subroutine
EXECUTE	Jump to address
!	Store integer to address
@	Fetch integer from address
R>	Pop from return stack
R@	Copy top of return stack
>R	Push on return stack
SWAP	Exchange top two integers on stack
OVER	Duplicate second integer on stack
2DROP	Discard two integers off stack
+	Add top two integers on stack
NOT	Complement top of stack
NEGATE	Negate top of stack
1-	Add -1 to top of stack
1+	Add 1 to top of stack

```
(makehead) BL
   20 ldi ret
(makehead) +!
   tx ldx add stx
   ret
(makehead) -
   com add 1 ldi add
   ret
(makehead) OR
   com pushr com
   popr and com ret
(makehead) ROT
   pushr pushr tx popr
   popr xt ret
(makehead) 2DUP
   pushs pushr pushr
   pushs tx popr xt popr
   ret
(makehead) 2!
   tx pushr stxp
   popr stx ret
(makehead) 2@
   tx ldxp ldx ret
(makehead) COUNT
   tx ldxp pushr xt
   popr ret

(makehead) DUP pushs ret
(makehead) DROP pops ret
(makehead) AND and ret
(makehead) XOR xor ret
(makehead) COM com ret

h forth_@

0 org
forth_' COLD >body forth_@ #,
0 #, 0 #, 0 #,

$24 org
$80 #,
0A #,
lasth forth_@ #,
           #,
lasth forth_@ #,
forth_' $INTERPRET >body forth_@ #,
forth_' QUIT >body forth_@ #,
0 #,
0 #,
lasth forth_@ #,
```

BL	Return $20
+!	Add second integer to address on top of stack
-	Subtract top of stack from second integer
OR	OR top two integers on stack
ROT	Rotate third integer to top of stack
2DUP	Duplicate top two integers on stack
2!	Store second and third integers as a double integer to the address on top of stack
2@	Fetch double integer from address on top of stack
COUNT	Read contents from address on top of stack; increment address
DUP	Duplicate top of stack
DROP	Discard top of stack
AND	AND top two integers on stack
XOR	XOR top two integers on stack
COM	1's Complement of top of stack

Initialize System Variables

When the eP32 powers up, the P register is cleared to 0, so we have to have some valid machine instruction at address 0 to boot up the eP32. The eForth boot up routine is the command COLD. Therefore, in memory location 0, we assemble a JMP COLD instruction.

Memory locations 1-$1F contain an interrupt vector table for interrupt services. However, no interrupt is expected in this eP32 system, and this area is cleared to 0. System variables are in the area between $20 and $2F. They contain vital information for the eP32 eForth system to work properly. Only the following system variables have to be initialized:

System Variable	Address	Initial Value	Function
'TIB	$24	$80	Pointer to Terminal Input Buffer.
BASE	$25	$0A	Number base for numeric conversions.
CONTEXT	$26	$7C1	Pointer to name field of last command in dictionary.
CP	$27	$7C3	Pointer to top of dictionary, first free memory location to add new commands. It is saved by "h forth_@" on top of the source code page.
LAST	$28	$7C1	Pointer to name field of last command.
'EVAL	$29	$4A0	Execution vector of text interpreter, initialized to point to $INTERPRET. It may be changed to point to $COMPILE in compiling mode.
'ABORT	$2A	$4D2	Pointer to QUIT command to handle error conditions.
tmp	$2B	$0	Scratch pad.
cpi	$2C	$0	Instruction slot counter for assembler.
cpw	$2D	$7C3	Pointer to top of dictionary, first free memory location to assemble machine instructions.

6.6 eP32 Simulator

An accurate and fast logic simulator is extremely valuable in designing and testing a new CPU. It is also very useful in separating hardware and software development, so that hardware and software can be developed simultaneously. This eP32 simulator served me well in the process of developing the eP32 CPU and its associated eForth system simultaneously.

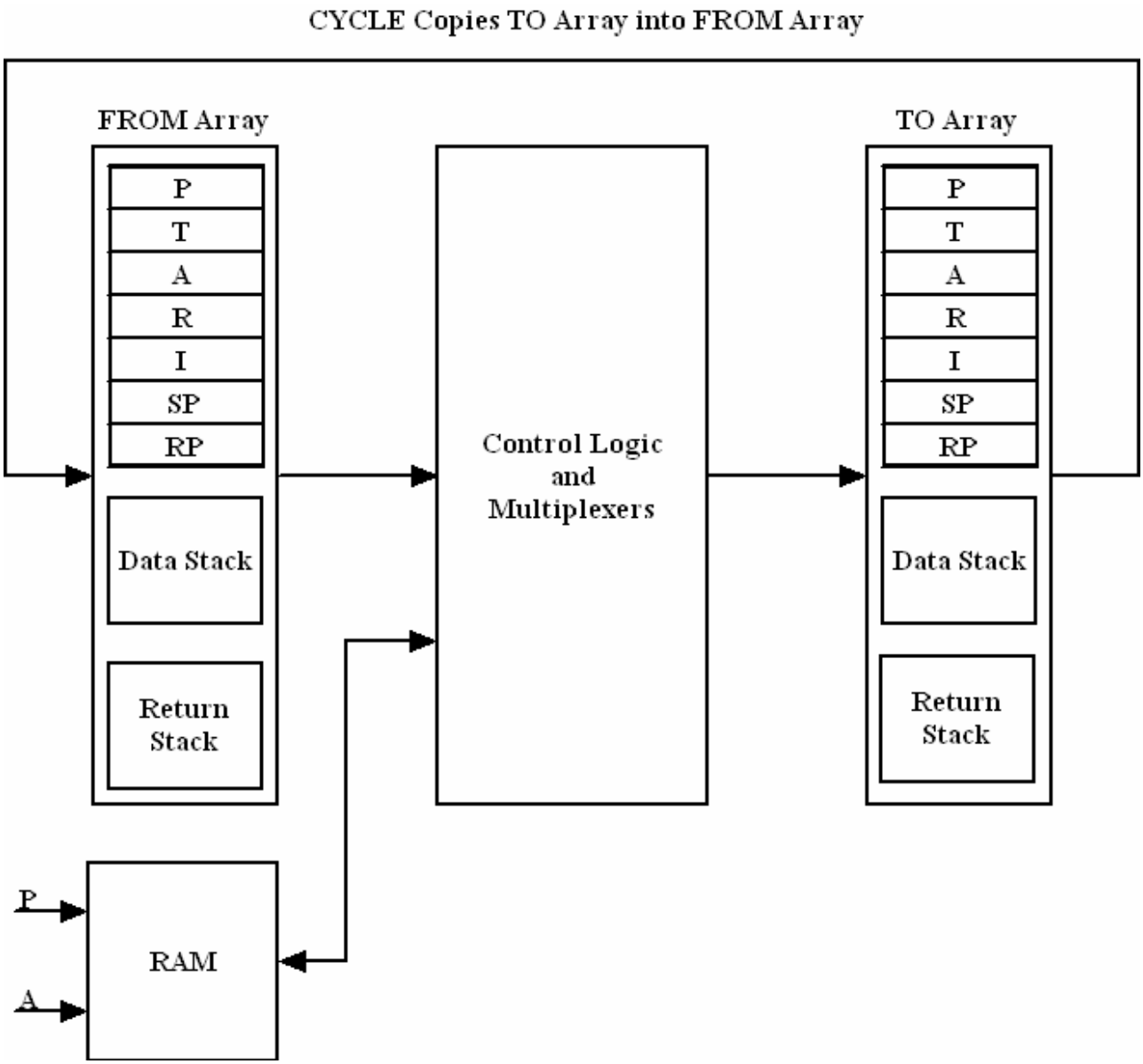

Figure 37. eP32 Simulator

This eP32 simulator faithfully replicates the logic behavior of the eP32 CPU on a cycle-by- cycle basis. The eP32 CPU is composed of a set of registers and two stacks. The registers and stacks latch input signals on the rising edge of the master clock. It is very simple to simulate this behavior logically in software.

The adder in the eP32 produces a 32-bit sum and a carry bit. To allow maximal

programming flexibility, the carry bit must be preserved in all registers and on stacks. Each register and all stack elements are represented by two 32-bit words. The first word contains the current value of the register, and the second word contains the carry bit associated with this value.

A large array, REGISTER, is opened to host these 64-bit double integers. It is divided in two banks: a FROM bank and a TO bank. The FROM bank contains current values of all registers and all stack elements. A machine instruction takes data in the FROM bank, modifies them, and writes updated data into the TO bank. The rising edge of the master clock copies the TO to the FROM bank, and thus simulates a machine instruction. Multiplexers in the eP32 are replaced by FORTH words that perform logic functions and update values from the FROM bank to the TO bank.

The Slot Machine, which fetches a program word from memory, and executes 5 machine instructions in this word, is simulated by a 32-bit counter. The least significant 3 bits in this counter steps through slots 0 to 5 in 6 clock cycles. Then this 3-bit field is cleared to zero and the upper 29-bit field is incremented. Therefore, the upper 29-bit field in this counter gives an accurate program word count.

The most interesting feature of this eP32 simulator is that it vectors KEY and EMIT commands to equivalent Windows functions “get” and “put”, so that the simulator can actually run eP32 eForth interactively on a Windows computer, and produces identical outputs as actual an eP32 microprocessor would do on a terminal. The simulator was proven to run identically to an actual eP32 microprocessor. This simulator can be used for software development, in place of a real eP32 microprocessor.

The source code of this simulator is in SIM32q.F. It is loaded at the end of META32q.F, which builds an eP32 eForth system in memory array “ram”. The simulator reads program words from this array and executes instructions contained in these program words.

The KEY and EMIT commands in the target eP32 system are patched so that eForth accepts characters from a PC keyboard and sends characters to the weFORTH console window on the PC screen. We add two machine instructions in the simulator: Instruction “get” (code $3E) receives a character from the PC and instruction “put” (code $3F) sends a character to the PC. Program word $3E11E79E contains these machine instructions: get/ret/nop/nop/nop, and is patched into the code field of KEY. Program word $3F11E79E contains these machine instructions: put/ret/nop/nop/nop, and is patched into code field of EMIT.

Once the KEY and EMIT commands are patched to do equivalent Windows functions, this simulator can actually run the eP32 eForth interactively, and it produces identical output as actual eP32 microprocessor would do on a terminal.

“forth_forget h” truncates the eForth dictionary back to where “h” was defined. It thus deletes words defined in the metacompiler, assembler, kernel, and target eP32. eForth is cleaned to a pristine state to host a new application, which is the eP32 simulator.

To manipulate double integers representing a value in registers and stacks, we need a set of ALU commands operating on double integers:

Command	Function
D+	Add top two double integers.
D-	Subtract top double integer from second double integer.
DNEGATE	Negate top double integer. 2's complement.
D2/	Shift double integer to right by 1 bit.
D2*	Shift double integer to the left by 1 bit.
LIMIT	Limit stacks depths are 256 levels.
RANGE	Limit program size to 32kB, the size of the 'RAM' array
CLOCK	A variable that has a 29-bit program word count field and a 3-bit SLOT field. The SLOT field sequences program word fetch and execution of up to 5 instructions in the program word.
BREAK	A variable holding breakpoint address.
(REGISTER)	A variable pointing either to the FROM bank or to the TO bank.
FROM	Switch register array to the FROM bank.
TO	Switch register array to the TO bank.
REGISTER	Base address of registers and stack arrays.

The eP32 CPU is paced by a single master clock. Registers, stacks, and memory contents are latched on the rising edge of the master clock. This latching action must be simulated accurately. The eP32 Simulator uses two register arrays, a FROM bank and a TO bank. Logic circuitry takes data from the FROM array and operates on them according to the current machine instruction, and stores results in the TO array. The rising edge of the master clock is simulated by copying the contents of the TO array to the FROM array, and then the system is ready for actions in the next clock cycle.

Registers and stacks are defined as pointers pointing into the REGISTER array:

Register	Function
P	Program counter
T	Accumulator, top item on data stack
S	Second item on data stack
R	Top of return stack
X	Address register
I	Instruction latch
I1	Machine instruction in slot1
I2	Machine instruction in slot2
I3	Machine instruction in slot3
I4	Machine instruction in slot4
I5	Machine instruction in slot5
RP	Return stack pointer
SP	Data stack pointer
RSTACK0	Origin of return stack
SSTACK0	Origin of data stack
RSTACK	Address of top of return stack
SSTACK	Address of top of data stack

```
HEX
3E11E79E forth_' KEY  >body forth_@ ram!
3F11E79E forth_' EMIT >body forth_@ ram!
forth_forget h
DECIMAL
: D+ ROT + >R UM+ R> + ;
: DNEGATE NEGATE >R NEGATE DUP IF -1 ELSE 0 THEN R> + ;
: D- DNEGATE D+ ;
: D2/ DUP 2/ >R 1 AND IF 2/ $80000000 OR ELSE 2/ $7FFFFFFF AND THEN
R> ;
: D2* 2* >R DUP $80000000 AND IF 2* R> 1 OR ELSE 2* R> THEN ;

$1F CONSTANT LIMIT ( stack depth )
$1FFF CONSTANT RANGE ( program memory size in words )
VARIABLE CLOCK ( slot is in the last 3 bits )
VARIABLE (REGISTER) ( where registers and stacks are )
VARIABLE BREAK

: REGISTER  (REGISTER) @ ;
: FROM  PAD (REGISTER) ! ;
: TO    PAD $600 + (REGISTER) ! ;

: P     REGISTER ;
: I     REGISTER 4 + ;
: I1    REGISTER 8 + ;
: I2    REGISTER 9 + ;
: I3    REGISTER 10 + ;
: I4    REGISTER 11 + ;
: I5    REGISTER 12 + ;
: RP    REGISTER 13 + ;
: SP    REGISTER 14 + ;
: T     REGISTER 16 + ;
: R     REGISTER 24 + ;
: X     REGISTER 32 + ;
: S     REGISTER 56 + ;
: RSTACK  RP C@ LIMIT AND 8 * REGISTER + $100 + ;
: SSTACK  SP C@ LIMIT AND 8 * REGISTER + $200 + ;

: CYCLE TO P FROM P $600 CMOVE 1 CLOCK +! ;

: JUMP  CLOCK @ 7 OR CLOCK ! ;

: RPUSH ( d -- , push d on return stack )
       FROM R 2@ RP C@ 1 + LIMIT AND TO RP C! RSTACK 2! R 2! ;

: RPOPP ( -- d , pop d from return stack )
       FROM R 2@ RSTACK 2@ RP C@ 1 - LIMIT AND TO RP C! R 2! ;

: SPUSH ( d -- , push d on data stack )
       FROM S 2@ SP C@ 1 + LIMIT AND TO SP C! SSTACK 2!
       FROM T 2@ TO S 2!
       TO T 2! ;

: SPOPP ( -- d , pop d from data stack )
       FROM T 2@
       FROM S 2@ TO T 2!
       FROM SSTACK 2@ SP C@ 1 - LIMIT AND TO SP C! S 2! ;
```

The Slot Machine paces the simulator through eP32 instructions stored in 'RAM' memory, just like the real eP32 CPU would do. Instead of using a single phase clock as master clock, we use a CLOCK variable as source of a multiple phase clock. The lowest three bits in CLOCK, Slot Counter, runs the slots in the slot machine. Its value indicates which slot is currently running. If it is 0, Slot0 is executed. If it is 1, Slot1 is executed. Etc. On the rising edge of the master clock, this slot counter is incremented. When slot count is 5, Slot5 is executed and the slot counter is reset to 0, so that next time Slot0 is executed.

JUMP also clears the Slot Counter to 0. JUMP is used by all transfer instructions to force the slot machine to enter slot0 on the rising edge of the next clock.

Command	Function
CYCLE	Simulate rising edge of master clock by incrementing CLOCK.
JUMP	Fetch next program word by forcing a 7 into Slot Counter in CLOCK. On the rising edge of the master clock, CLOCK is incremented and clears Slot Counter to 0. The upper 29-bit field in CLOCK is incremented, indicating that a new word is fetched from memory. Thus the upper 29 bits in CLOCK keeps an accurate count of eP32 words that have been executed.
RPUSH	Push double integer d on return stack.
RPOPP	Pop return stack and leave double integer on system stack.
SPUSH	Push double integer d on data stack.
SPOPP	Pop data stack and leave double integer on system stack.

"continue" simulates functions performed in slot0 in the Slot Machine, which fetches the next program word from memory and stores it in instruction register I. Machine instructions in slot1 to slot5 are extracted to operate a decoder, which generates control signals for all components in the eP32.

"continue" also increments the P register, and copies machine instructions in slot1 to slot5 to instruction registers I1-I5.

To execute a machine instruction, the simulator takes current values in registers and stacks in the FROM bank, computes desired new values, and deposits them back in registers and stacks in the TO bank. On the rising edge of the master clock, which is simulated by command CYCLE, the contents of the TO bank are copied to the FROM bank. Machine instructions are defined as commands in this simulator, and they read values in the FROM bank, make necessary changes, and store new values in the TO bank.

As registers and stacks are represented in double integers, math operations are performed using double integer math commands defined at the beginning of the simulator. They are D+, D-, DNEGATE, D2*, and D2/.

```
: continue
       FROM P @ DUP 1+ TO RANGE AND P !
       ram@ DUP I !
       64 /MOD SWAP I5 C!
       64 /MOD SWAP I4 C!
       64 /MOD SWAP I3 C!
       64 /MOD SWAP I2 C!
       63 AND I1 C!
       ;

: nop   JUMP ;
: ei    ;
: di    ;
: bra   I @ TO RANGE AND P ! JUMP ;
: ret   RPOPP DROP TO RANGE AND P !
       JUMP ;
: bn    SPOPP DROP 0< ( branch on sign )
       IF bra ELSE JUMP THEN ;
: bc    SPOPP SWAP DROP ( branch on carry )
       IF bra ELSE JUMP THEN ;
: bz    SPOPP DROP ( branch on zero )
       IF JUMP ELSE bra THEN ;
: call  FROM P @ 0 RPUSH bra ;
: next  FROM R 2@ DROP
       IF ELSE RPOPP 2DROP JUMP EXIT THEN ( exit loop )
       FROM R 2@ DROP 1- 0 TO R 2! ( decrement R )
       FROM bra ;
: times FROM R 2@ DROP
       IF ELSE JUMP EXIT THEN ( exit loop )
       R 2@ 1 0 D- TO R 2! ( decrement R )
       FROM -1 P +! TO -1 P +! ;
: pushr SPOPP RPUSH ;
: dupr  FROM R 2@ SPUSH ;
: popr  RPOPP SPUSH ;
: andd  SPOPP DROP TO T 2@ DROP AND 0 T 2! ;
: xorr  SPOPP DROP TO T 2@ DROP XOR 0 T 2! ;
: com   FROM T 2@ DROP -1 XOR 0 TO T 2! ;
: add   SPOPP DROP 0 TO T 2@ DROP 0 D+ TO 1 AND T 2! ;
: mul   FROM X 2@ DROP 1 AND
       IF S 2@ T 2@ D+
       ELSE T 2@ THEN 1 AND
       2DUP D2/ TO T 2!
       DROP 1 AND >R
       FROM X 2@ DROP 2/ $7FFFFFFF AND R> IF $80000000 OR THEN TO 0
X 2! ;
: div   FROM S 2@ DROP 0 T 2@ DROP 0 D+
       1 AND DUP >R DUP
       IF ELSE 2DROP T 2@ THEN
       D2* ( diff) 1 AND X 2@ DROP $80000000 AND IF 1 0 D+ THEN TO T
2!
       FROM X 2@ DROP 2* R> IF 1+ THEN TO 0 X 2! ;
: shr   FROM T 2@ DROP 2/ 1 TO T 2! ;
: shl   FROM T 2@ D2* 1 AND TO T 2! ;
: rr8   FROM T 2@ DROP DUP 7 FOR D2/ NEXT DROP 0 TO T 2! ;
: ldi   FROM P @ 1+ TO RANGE AND P !
       FROM P @ RANGE AND ram@ 0 SPUSH ;
```

nop	No operation.
ei	Enable interrupt.
di	Disable interrupt.
bra	Jump to address contained in current instruction.
ret	Return from a subroutine to main program. Pop return address from return stack and store it in P.
bn	If T<0 is set, jump to address contained in current instruction; else continue.
bc	If Carry is set, jump to address contained in current instruction; else continue.
bz	If T=0, jump to address contained in current instruction; else continue.
call	Push address in P on R stack, and jump to address contained in current instruction; else continue.
next	If R is not 0, jump to address contained in current instruction, and decrement R by 1; else pop R stack and continue.
times	Micro loop. Similar to “next”, except repeating instructions in current program word.
pushr	Push T onto R stack. Pop S stack to T.
dupr	Push T onto S stack. Dup R to T.
popr	Push T onto S stack. Pop R stack to T.
andd	Pop S stack and AND it to T.
xorr	Pop S stack and XOR it to T.
com	Complement T (1’s complement).
addd	Pop S stack and add it to T.
mul	Multiplication step. If X(0)=1, add S to T, otherwise T is not changed. Shift T:X pair right by 1 bit.
div	Division step. If T+S produces a carry, add S to T, otherwise T is not changed. Shift T:X pair left by 1 bit. Shift carry into X(0).
shr	Shift T right by 1 bit.
shl	Shift T left by 1 bit.
rr8	Rotate T right by 8 bits.
ldi	Push T on S stack, read memory word pointed by P into T. Increment P by 1.
pushs	Push T on S stack.
xt	Push T on S stack. Copy X to T.
pops	Pop S stack to T.
overr	Push T on S stack. Copy original contents of S to T.
tx	Copy T to X. Pop S stack to T.
ldx	Push T on S stack, read memory word pointed by X into T.
ldxp	Push T on S stack, read memory word pointed by X into T. Increment X by 1.
ldrp	Push T on S stack, read memory word pointed by R into T. Increment R by 1.
stx	Store T into memory pointed by X. Pop S stack to T.
stxp	Store T into memory pointed by X. Increment X by 1. Pop S stack to T.
strp	Store T into memory pointed by R. Increment R by 1. Pop S stack to T.

We want the simulator to run the eP32 eForth system. The real eP32 microprocessor talks to a host computer through a UART serial port. Normally we use HyperTerminal in Windows to interact with the eP32. To simulate interaction between the eP32 and HyperTerminal, we have to hijack the output of EMIT and send it to the weFORTH console window, and intercept keyboard strokes from the computer keyboard and feed them to KEY in eForth. These two functions are implemented in the simulator by creating two special machine instructions, “get” and “put”, which use machine codes $3E and $3F, respectively.

“get” and “put” are patched into the code fields of KEY and EMIT in the memory array “ram” so that when the simulator executes EMIT, a character is displayed on the weFORTH console, and when KEY is executed, an ASCII character is accepted from the keyboard. With “get” and “put”, the simulator runs the eP32 eForth system identically like the eP32-HyperTerminal system.

get	Force simulator to get a character from keyboard under Windows.
put	Force simulator to send a character to weFORTH console window.

“execute” is a giant case statement that gets “code” from the top of the stack and selects the proper commands to simulate a machine instruction in this emulator. Since weForth did not bother to define case structure and associated control commands, we just use lots of IF-THEN structures to emulate a case structure. “code” is duplicated on the stack and compared with consecutive machine code. If a match is found, the corresponding command is executed to simulate that machine instruction. After that, EXIT is executed, and “execute” is terminated. Further comparisons are not necessary.

If “code” does not match a valid machine code, we have a very serious problem. Either the eForth program has a bug, or the eP32 simulator has a bug. This simulator is aborted. The offending “code” is displayed with an error message. The eForth system returns to its default text interpreter, and you can type in eForth commands to find and correct this bug.

```
: pushs FROM T 2@ SPUSH ;
: xt    FROM X 2@ SPUSH ;
: pops  SPOPP 2DROP ;
: overr FROM S 2@ SPUSH ;
: tx    SPOPP TO X 2! ;
: ldx   FROM X 2@ DROP RANGE AND ram@ 0 SPUSH ;
: ldxp  ldx
       FROM X 2@ 1 0 D+ 1 AND TO X 2! ;
: ldrp  FROM R 2@ DROP RANGE AND ram@ 0 SPUSH
       FROM R 2@ 1 0 D+ 1 AND TO R 2! ;
: stx   SPOPP DROP FROM X 2@ DROP RANGE AND ram! ;
: stxp  stx
       FROM X 2@ 1 0 D+ 1 AND TO X 2! ;
: strp  SPOPP DROP FROM R 2@ DROP RANGE AND ram!
       FROM R 2@ 1 0 D+ 1 AND TO R 2! ;
: get   KEY DUP $1B = ABORT" done"
       0 SPUSH ret ;
: put   SPOPP DROP $7F AND EMIT ret ;

HEX
: execute ( code -- )
       DUP 00 = IF DROP bra   EXIT THEN
       DUP 01 = IF DROP ret   EXIT THEN
       DUP 02 = IF DROP bz    EXIT THEN
       DUP 03 = IF DROP bc    EXIT THEN
       DUP 04 = IF DROP call  EXIT THEN
       DUP 05 = IF DROP next  EXIT THEN
       DUP 06 = IF DROP times EXIT THEN
\       DUP 07 = IF DROP di    EXIT THEN
       DUP 08 = IF DROP ldrp  EXIT THEN
       DUP 09 = IF DROP ldxp  EXIT THEN

       DUP 0A = IF DROP ldi   EXIT THEN
       DUP 0B = IF DROP ldx   EXIT THEN
       DUP 0C = IF DROP strp  EXIT THEN
       DUP 0D = IF DROP stxp  EXIT THEN
       DUP 0E = IF DROP rr8   EXIT THEN
       DUP 0F = IF DROP stx   EXIT THEN
       DUP 10 = IF DROP com   EXIT THEN
       DUP 11 = IF DROP shl   EXIT THEN
       DUP 12 = IF DROP shr   EXIT THEN
       DUP 13 = IF DROP mul   EXIT THEN
       DUP 14 = IF DROP xorr  EXIT THEN
       DUP 15 = IF DROP andd  EXIT THEN
       DUP 16 = IF DROP div   EXIT THEN
       DUP 17 = IF DROP add   EXIT THEN
       DUP 18 = IF DROP popr  EXIT THEN
       DUP 19 = IF DROP xt    EXIT THEN
       DUP 1A = IF DROP pushs EXIT THEN
       DUP 1B = IF DROP overr EXIT THEN
       DUP 1C = IF DROP pushr EXIT THEN
       DUP 1D = IF DROP tx    EXIT THEN
       DUP 1E = IF DROP nop   EXIT THEN
       DUP 1F = IF DROP pops  EXIT THEN
       DUP 3E = IF DROP get   EXIT THEN
       DUP 3F = IF DROP put   EXIT THEN
       . ABORT" :Illegel instruction" ;
```

Here are the commands that run Slot Machine, and show the contents of pertinent registers and stacks. Originally, I thought of implementing a set of break points to allow user the freedom to break execution at a number of different memory locations. Eventually, I realized that only one break point is necessary and a simple 'GO' command is sufficient. This is the G command show below.

Command	Function
.stack	Display the contents of a stack.
.sstack	Display the contents of data stack.
.rstack	Display the contents of return stack.
.registers	Display the contents of all the relevant registers.
S	Show all the registers and stacks at this cycle.
sync	Execute the current machine instruction using CLOCK to determine which slot is being executed. CLOCK points to one of the routines in SYNC-TABLE, which contains the following entries: CONTINUE, fetch next program word SYNC1, execute instruction in I1 SYNC2, execute instruction in I2 SYNC2, execute instruction in I2 SYNC3, execute instruction in I3 SYNC4, execute instruction in I4 SYNC5, execute instruction in I5
C	Run one clock cycle and display all registers and stacks.
reset	Clear the REGISTER array, simulating hardware reset.

"C" is the single stepper in simulator. It runs the Slot Machine for one cycle, and displays all registers and stacks. This is the most useful command to debug the eP32 in the early development stage. You can see all data in all registers and stacks. In the eP32 eForth system, the first command executed is COLD, which executes a diagnostic word, DIAGNOSE. DIAGNOSE runs simple tests on most machine instructions. By single stepping through DIAGNOSE, you can validate most machine instructions. If all tests in DIAGNOSE run successfully, it is very likely the eP32 will run correctly in the FPGA.

"reset" clears the REGISTER array, and initializes the simulator to run at memory location 0.

This simulator has a very simple text-based user interface. The most used commands are:

Command	Stack Effects	Function
G	--	Run and stop at address given on FORTH stack. This is a very efficient way to set breakpoints and then run till a breakpoint is triggered. It allows the user to execute a large portion of the program and stop only at a specified location.
PUSH	n --	Push a new integer into the T register and data stack.

POP	--	Discard contents in T and pop data stack back into T.
D	--	Display memory starting at address in P.
M	a --	Dump 128 words in memory using “show” command.
RUN	--	Continue stepping with any key, terminated by ESC.
P	a --	Start simulating at the address on stack.

This simulator is most effective in debugging short sequences of program words to verify that the sequences are executed correctly. After eP32 machine instructions are verified, use the G command to execute a long stretch of program and break only at a specified location. This allows large segments of programs to be tested. If the simulator runs forever and cannot reach the break point you specified, you can stop the G command by hitting a key on the keyboard to terminate it.

When weForth runs the metacompiler to compile an eForth system for the eP32, it displays names and code field addresses of all commands compiled into the target image. The display is a symbol table. You can look up a command and find its code field address. The code field addresses are the best place to set your break point. To debug a command, find its code field address and enter it with the G command. The simulator will break at the beginning of this command, and you can use the C command to single step through it.

Typing lots of “C” commands is tedious. The RUN command lessens your typing chore. After executing RUN, the simulator displays registers and stacks and pauses. Pressing any key will single step Slot Machine for one cycle. You can run many steps easily this way. When you want to stop RUN, press the ESC key.

To examine memory, type an address followed by the “M” command. It will display 128 words of memory starting from that address. The “D” command displays 8 program words starting at this address.

If you want to start debugging at a particular address, type the address followed by the “P” command. This address is stored in the program counter register, P, and “C” or “RUN” commands will single step words starting at this memory address.

If you want to change the data stack to run simulation with the data you want on the stack, use “PUSH” and “POP” commands. Type a number followed by “PUSH”, and this number is pushed on the data stack in the simulator. You can enter as many numbers on stack as you like in this way. If you want to pop a number off the data stack, type “POP”.

The above commands allow you to set up the eP32 in the simulator exactly the way you want before running simulation.

The HELP command displays a help screen to remind you of simulation commands and arguments they need on the data stack.

```
: .stack ( add # ) FOR AFT DUP 2@ DROP U. 8 - THEN NEXT DROP CR ;
: .sstack ." S:" T 2@ IF ." C" THEN U.
      S 2@ DROP U. SSTACK SP C@ .stack ;
: .rstack ." R:" R 2@ DROP U. RSTACK RP C@ .stack ;
: .xstack ." X:" X 2@ DROP U. ;
: .registers ."  P=" P @ . ."  I=" I @ U.
      ."  I1=" I1 C@ . ."  I2=" I2 C@ .
      ."  I3=" I3 C@ . ."  I4=" I4 C@ .
      ."  I5=" I5 C@ . CR ;
: S  CR ." CLOCK=" CLOCK @ . .registers
      .sstack .rstack .xstack ;

: sync  CLOCK @ 7 AND
      DUP 0 = IF continue DROP EXIT THEN
      DUP 1 = IF   I1 C@ execute DROP EXIT THEN
      DUP 2 = IF   I2 C@ execute DROP EXIT THEN
      DUP 3 = IF   I3 C@ execute DROP EXIT THEN
      DUP 4 = IF   I4 C@ execute DROP EXIT THEN
      DUP 5 = IF   I5 C@ execute           THEN
      DROP JUMP ;
: C    sync CYCLE S ;
: reset FROM P $C00 0 FILL 0 CLOCK ! ;
reset

: G    ( addr -- )
      CR ." Press any key to stop." CR
      BREAK !
      BEGIN sync P @ BREAK @ =
            IF CYCLE C EXIT
            ELSE CYCLE
            THEN
            ?KEY
      UNTIL ;
: PUSH  ( n ) pushs TO 0 T 2! ;
: POP   pops ;

: D    P @ 1- four four ;
: M    show ;
: RUN   CR ." Press ESC to stop." CR
      BEGIN C KEY 1B = UNTIL ;
: P    DUP FROM RANGE AND P ! TO RANGE AND P ! ;

: HELP  CR ." eP32 Simulator, copyright eForth Group, 2000"
      CR ." C: execute next cycle"
      CR ." S: show all registers"
      CR ." D: display next 8 words"
      CR ." addr M: display 128 words from addr"
      CR ." addr P: start execution at addr"
      CR ." addr G: run and stop at addr"
      CR ." RUN: execute, one key per cycle"
      CR ;
```

Conclusion

In early 1990's, when I worked with Chuck Moore on the MuP21 chip, he was daydreaming one afternoon, and said something like this: "I wish that I had a machine like a microwave oven on my kitchen table. I would put in a piece of silicon and turn on the power switch. After half a hour, I would open the door, and there is my chip."

With LatticeXP2-5E FPGA chip on Brevia Kit, I am practising Chuck's dream now, on my desk.

You can practise Chuck's dream also. You can design and produce your own microprocessor. You can write your own programming language and operating system. All you have to do is to sit back, think hard, and find a good application that you can sell a million chips.

In the FORTH programming language and in the designs of FORTH microprocessors, Chuck Moore reduced computer software and computer hardware to their simplest forms, which can be understood, reproduced, and improved by ordinary people like us. You do not have to be Intel or Microsoft to make computers and to solve application problems.

“Yes, we can! Yes, we can! Yes, we can!”

Appendix A: eP32 Instruction Set

Here I will present formal definitions of all eP32 instructions. They begin with the assembly mnemonics and a name, followed by their code, usage, stack effects, and effects on the carry bit. These attributes are presented in a table. Then there is a detailed description of the instruction's function followed by some coding examples. Usage rows show how an instruction appears in a 32-bit program word, using following notations:

Notation	Representation
00	Highest two bits, not used
iiiiii	Current instruction code in binary
cccccc	6 bit instruction code
nnnnnn	6 bit data
aaaaaa	6 bit address
xxxxxx	6 don't care bits

The stack effect row shows how this instruction affects the data stack, return stack, and sometimes the X register. Stack effects are shown in the following style:

Items before execution – items after execution

Items are identified using the following notation:

Notation	Representation
n	a general 32-bit integer
a	a 32-bit address
f	a logic flag, true=-1, false=0

If an instruction changes the return stack and the X register, these effects are added to the data stack effects separated by colons:

n1 n2 – n3 n4 ; R: -- n ; X: -- n

The carry row shows how the carry bit is changed by the instruction.

Coding examples are often taken from the kernel of the eForth system in the files KERN32q.F and EF32q.F. Code fragments are generally shown in machine code format. Complete definitions of code commands are shown in eForth assembly format and FORTH compound commands are shown in FORTH format. You are encouraged to read these files and examine these examples in their original context.

ADD Addition

Code:	23
Usage	Short Instruction
Stack Effects	(n1 n2 -- n1+n2)
Carry	Change according to n1+n2

Function:

Pop S from the data stack and add it to the T register.

Coding Example:

The primitive addition word in eForth is thus defined:

```
CODE UM+  ( n n - n carry )
   add pushs
   ifnc pushs pushs xor ret
   then
   1 ldi ret
: NEGATE ( n -- -n )  com 1 ldi add ;
: 1- ( a -- a )  -1 ldi add ;
: 1+ ( a -- a )   1 ldi add ;
: +! ( n a -- )  tx ldx add stx ;
: - ( w w -- w )   com add 1 ldi add ;
```

AND Bitwise AND

Code:	21
Usage	Short Instruction
Stack Effects	(n1 n2 -- n3)
Carry	AND of bits n1(32) and n2(32)

Function:

Pop S from the data stack and bitwise AND it to the T register. All 33 bits in T are affected.

Coding Example:

To generate a 0 in the T register:

```
    DUP DUP COM AND
```

To convert a numeric digit to its corresponding ASCII code:

```
:: DIGIT ( u -- c )
  9 LIT OVER < 7 LIT AND +
  ( CHAR 0 ) 30 LIT +
;;
```

BC Branch on Carry

Code:	3
Usage	00 000011 aaaaaa aaaaaa aaaaaa aaaaaa
Stack Effects	(n --)
Carry	Restored from data stack

Function:

Conditionally branch to the 24-bit address in the bit field 23-0 in the current 16M word page of memory, if the Carry flag (Bit 32 of T) is set. It must be in slot1 of a program word. The current value in the T register is destroyed and the data stack is popped back to T. This instruction is different from BRA, which does not change the data stack or T.

Coding Example:

The negative flag T(31) is shifted into carry T(32). BC compiled by IFNC tests this.

```
CODE ABS ( n -- +n )
   pushs shl
   ifnc ret then
   negate ret
```

BRA Branch Always

Code:	0
Usage	00 000000 aaaaaa aaaaaa aaaaaa aaaaaa
Stack Effects	None
Carry	No change

Function:
Branch to the 24-bit address in bit field 23-0 in the current 16M word page of memory. It must be in slot1 of a program word. BRA is compiled by ELSE, REPEAT and AGAIN to construct branch and loop structures.

Restriction:

This instruction allows the program to be redirected to any location within a 16M word page of memory. It does not cross page boundaries. To jump to locations outside of a memory page, one has to push a target address onto the return stack and execute the RET instruction to cause a long jump. This restriction also applies to CALL, BZ, BC, and NEXT. See also RET.

Coding Example:

To delay 50 or 100 micro seconds:

```
CODE 50us
2 ldi skip
CODE 100us
1 ldi
then
sta -138 ldi
begin lda add
-until
drop
ret
```

SKIP compiles an unconditional branch, BRA, to THEN, to let the routine ‘50us’ share a delay loop with the routine ‘100us’.

BZ Branch on Zero

Code:	2
Usage	00 000010 aaaaaa aaaaaa aaaaaa aaaaaa
Stack Effects	(n --)
Carry	Restored from data stack

Function:

Conditionally branch to the 24-bit address in the bit field 23-0 in the current 16M word page of memory, if the T register contains a 0. It must be in slot1 of a program word.

The T register is destroyed and the data stack is popped back to T. This instruction is different from BRA, which does not change the data stack or T. BZ is compiled by IF, WHILE and UNTIL to construct branch and loop structures.

Coding Example:

```
CODE ?DUP ( w -- w w | 0 )
   pushs
   if pushs ret then
   ret
```

CALL Call Subroutine

Code:	4
Usage	00 000100 aaaaaa aaaaaa aaaaaa aaaaaa
Stack Effects	(-- ; R: -- a)
Carry	No change

Function:

Call a subroutine whose address is in bit field 23-0 in the current 16M word page of memory. It must be in slot1 of a program word.

The address of the next program word is pushed onto the return stack. When a return instruction in a subroutine is encountered, this address is popped off of the return stack back to the program counter and the next program word is executed to resume the execution sequence interrupted by the subroutine call.

Restriction:

This instruction allows the program to call any subroutine within the current 16M word page of memory. It does not cross page boundaries.

Coding Example:

All compound FORTH commands are compiled as subroutine calls. This is the most efficient way to build program lists in FORTH.

```
:: HERE ( -- a ) CP @ ;;
:: PAD ( -- a ) CP @ 100 LIT + ;;
:: TIB ( -- a ) 'TIB @ ;;
```

COM Bitwise Complement

Code:	16
Usage	Short Instruction
Stack Effects	(n – 1-n)
Carry	Reset to 0Complement of T(32)

Function:

Complement all 33 bits in the T register. It is a one's complement operation.

Coding Example:

To generate a 0 in the T register:

```
DUP DUP COM AND
```

To generate a -1 in the T register:

```
DUP DUP COM XOR
```

The first step is to make two copies of T. The topmost copy is complemented and then ANDed or XORed into second copy of T. All bits are cleared or set, and the result is a 0 or a -1 in T.

```
: NOT ( w -- w ) com ;
: NEGATE ( n -- -n )  com 1 ldi add ;
```

DIV Divide Step

Code:	22
Usage	Short Instruction
Stack Effects	(n1 n2 -- n1 n3)
Carry	Bit T(31) or Bit 31 from adder

Function:

Conditionally add the S register onto the data stack to the T register if the carry bit from addition is 1. If carry is 0, the T register is not modified. The T-X register pair is then shifted to the left by one bit. Carry is shifted into X(0).

This DIV instruction is useful as a divide step to implement a fast software division routine. Repeating this instruction 33 times will divide the T-X pair by S. The quotient is in X and the remainder is in T.

Coding Example:

Divide a 64-bit positive integer by a positive 31-bit divisor. A negated divisor is in S. The 64-bit dividend is in the T-X register pair.

```
CODE /MOD ( n n -- r q )
   com 1 ldi add pushr
   tx popr 0 ldi
   then
   div div div div
   div div div div
   div div div div
   div div div div
   div div div div
   div div div div
   div div div div
   div div div div
   div 1 ldi xor shr
   pushr pops popr xt
   ret
```

DROP Discard T Register

Code:	31
Usage	Short Instruction
Stack Effects	(n --)
Carry	Restore from data stack

Function:

Pop S from the data stack and store it in the T register. The original contents in the T register are lost. In assembler, DROP has an alias, ‘pops’.

Coding Example:

```
: DROP ( w w  -- )   pops ;
: 2DROP ( w w  -- ) pops pops ;
```

DUP Duplicate T Register

Code:	26
Usage	Short Instruction
Stack Effects	(n -- n n)
Carry	No change

Function:

Duplicate the T register and push it onto the data stack. In assembler, DUP has an alias, ‘pushs’.

Coding Example:

Create 0 in T `DUP DUP XOR AND`
Create -1 in T `DUP DUP XOR COM`
Decrement T `DUP DUP XOR COM ADD`

```
CODE 0< ( n - f )
   shl ifnc pushs pushs xor ret
   then
   -1 ldi ret
```

EI Enable Interrupts

Code:	6
Usage	Short Instruction
Stack Effects	None
Carry	No change

Function:

Enable external interrupts through the INTERRUPT(0-4) pins. When the eP32 is powered up, external interrupts are disabled. After EI is executed, the CPU will respond to external interrupts. Interrupt pins are sampled in slot0. If any of the 5 interrupt pins is pulled high, the CPU will force a subroutine call to an address between 1 and 31 according to the bit pattern sampled in INTERRUPT(0-4). Further interrupts are disabled, until another EI is executed.

Before executing EI, the system must write valid addresses of interrupt service routines into the interrupt vectors from locations 1 to 31, so that the system can respond correctly to simultaneous real time interrupts from 5 external devices.

LDI Load Immediate

Code:	10
Usage	Short Instruction followed by a 32-bit literal value
Stack Effects	(-- n)
Carry	Reset to 0

Function:

Fetch the contents of the next program word and push that number onto the data stack. The program counter, PC, is incremented, passing the next program word. This instruction allows a program to enter numbers (literals) onto the data stack at run time. It also resets the carry flag (Bit 32) in the T register.

Coding Example:

Push 1 2 3 4 on data stack:

```
    LDI LDI LDI LDI
    1
    2
    3
    4
CODE = ( w w -- t )
   xor
   if pushs pushs xor ret then
   -1 ldi ret
```

LDX Load from X Register

Code:	11
Usage	Short Instruction
Stack Effects	(-- n)
Carry	Reset to 0

Function:

Fetch the contents of a memory location whose 32-bit address is in the X register and push that number onto the data stack. The address in the X register is not modified.

This fetch instruction is different from the @ instruction in FORTH, which uses the address on top of the data stack.

This instruction also resets the carry flag (Bit 32) in the T register.

Coding Example:

```
: @  ( a - n ) tx ldx ;
: 2@ ( a -- d ) tx ldxp ldx ;
```

LDXP Load from X Register, Auto-Incrementing

Code:	9
Usage	Short Instruction
Stack Effects	(-- n ; X: a – a+1)
Carry	reset to 0

Function:

Fetch the contents of a memory location whose 32-bit address is in the X register and push that number onto the data stack. The address in the X register is then incremented to facilitate accessing the next memory location. It is most useful in reading values from an array in memory.

This fetch instruction is different from the @ instruction in FORTH, which uses the address on top of the data stack.

This instruction also resets the carry flag (Bit 32) in the T register.

Coding Example:

```
: 2@ ( a -- d ) tx ldxp ldx ;
```

MUL Multiply Step

Code:	19
Usage	Short Instruction
Stack Effects	(n1 n2 – lo hi)
Carry	Reset to 0Change to T(31) or sum(31)

Function:

Conditionally add the S register on the data stack to the T register if the lowest bit in the X register, X(0), is 1. If X(0) is 0, the T register is not modified. The T-X register pair is then shifted to the right by one bit.

This MUL instruction is useful as a multiply step in implementing a fast software multiplication routine. Repeating this instruction 32 times will multiply X and S and produces a 64-bit product in the T-X register pair. If the T register is not initialized to 0, its contents are added to the product.

Coding Example:

Multiply two 32-bit unsigned integers. Multiplicand is in X. Multiplier is in S.

```
CODE UM* ( u u -- ud )
   tx 0 ldi
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   mul mul mul mul
   pushr pops xt popr
   ret
```

The 32-bit product is in the T-X register pair. The multiplicand in S is preserved.

NEXT Loop Back

Code:	5
Usage	00 000101 aaaaaa aaaaaa aaaaaa aaaaaa
Stack Effects	(-- ; R: n – n-1 if n is not 0, n – if n=0)
Carry	No change

Function:

If the top of the return stack, R, is not zero, loop to the 24-bit address in bit field 23-0 in the current 16M word page of memory. R is decremented by 1. If R is 0, pop the return stack, terminate the loop, and continue executing the next program word. It must be in slot1 of a program word. NEXT is re-defined in assembler to terminate a loop structure by assembling a NEXT instruction.

Coding Example:

```
:: CMOVE ( b b u -- )
   FOR AFT
      over c@ over c!
      >R 1+ R> 1+
   THEN NEXT 2DROP ;;
:: FILL ( b u c -- )
   SWAP FOR SWAP AFT
      2DUP c! 1+
   THEN NEXT 2DROP ;;
```

NOP No Operation

Code:	30
Usage	Short Instruction
Stack Effects	(--)
Carry	No change

Function:

No operation. This instruction forces the execution sequencer to state slot0, and causes the next program word to be fetched and executed. All instructions in the current program word following NOP are ignored. In assembler, NOP is automatically padded into a program word to fill unused slots.

OVER Duplicate S Register

Code:	27
Usage	Short Instruction
Stack Effects	(n1 n2 – n1 n2 n1)
Carry	Restore from S register

Function:

Push the T register onto the data stack. Copy the original contents of S to T.

Coding Example:

```
:: 2DUP OVER OVER ;;
```

POP Pop Return Stack

Code:	24
Usage	Short Instruction
Stack Effects	(-- n ; R: n --)
Carry	Restore from return stack

Function:

Pop the R register on the return stack to the T register. The original contents in T are pushed onto the data stack.

Coding Example:

Exchanging X and T `STA PUSH LDA POP`
Exchanging X and R `STA POP LDA PUSH`
Increment T by 4 `STA LDP DROP LDA`
Decrement T by 4 `DUP DUP XOR COM ADD`

```
:: CMOVE ( b b u -- )
   FOR AFT over c@ over c!
      >R 1+ R> 1+
   THEN NEXT 2DROP ;;
```

PUSH Push Return Stack

Code:	28
Usage	Short Instruction
Stack Effects	(n -- ; R: -- n)
Carry	Restore from data stack

Function:

Pop S from the data stack and store it to the T register. The original contents in the T register are pushed onto the return stack.

Coding Example:

```
: 2DUP ( w1 w2 -- w1 w2 w1 w2 )
   over over
   ;
: ROT ( w1 w2 w3 -- w2 w3 w1 )
   pushr pushr tx popr
   popr xt ;
```

RET Return from Subroutine

Code:	1
Usage	Short Instruction
Stack Effects	(-- ; R: a --)
Carry	No change

Function:

Pop the top of the return stack into the program counter, P, and thus resume the execution sequence interrupted by the last CALL instruction. Besides terminating a subroutine, this instruction may be used to execute a long jump to a location outside of the current memory page. This instruction can be placed in any slot of a word. Instructions before RET are executed. Instructions following RET are ignored.

Coding Example:

In the Subroutine Threading Model, RET is used to terminal all code commands and colon commands. The word ";" simply compiles a RET to terminate a FORTH word.

```
CODE 0< ( n - f )
   shl ifnc pushs pushs xor ret
   then  -1 ldi ret
CODE UM+  ( n n - n carry )
   add pushs
   ifnc pushs pushs xor ret
   then  1 ldi ret
```

RR8 Rotate Right by 8 Bits

Code:	14
Usage	Short Instruction
Stack Effects	(n1 - n2)
Carry	No change

Function:

Rotate T to the right by 8 bits. The lowest 8 bits are moved to the highest 8 bits. This instruction is very useful in extracting bytes from a 32-bit integer in the T register, and to pack bytes into T.

Coding Example:

```
:: wupper ( w -- w' ) \ convert 4 bytes to uppercase
   3 LIT FOR
      DUP FF LIT AND 61 LIT 7B LIT WITHIN
      IF FFFFFF5F LIT AND THEN
      RR8
   NEXT
   ;;
```

SHL Shift Left

Code:	17
Usage	Short Instruction
Stack Effects	(n -- 2n)
Carry	Change to T(31)

Function:

Shift all lower 32 bits in the T register to left by 1 bit. The lowest Bit, T(0), is set to 0.

Coding Example:

Multiply T by 3: `DUP SHL NOP NOP ADD`
Multiply by 5: `DUP SHL SHL DOP ADD`
Multiply by 6: `SHL DUP SHL NOP ADD`

SHL allows the negative bit, T(31), to be tested as the carry bit T(32):

```
CODE CELL*  SHL SHL RET
CODE 0< ( n - f )
SHL
-IF -1 LDI RET
THEN
DUP XOR ( 0 LDI )
RET
```

SHR Shift Right

Code:	18
Usage	Short Instruction
Stack Effects	(n -- n/2)
Carry	Reset to 0

Function:

Shift the lower 32 bits in the T register right by one bit. Bit T(0) is lost. The sign bit, T(31), is preserved. The carry bit, T(32), is cleared.

Coding Example:

```
CODE 4/  SHR SHR RET
```

STX Store with X Register

Code:	15
Usage	Short Instruction
Stack Effects	(n --)
Carry	Restore from data stack

Function:

Store T into the memory location whose 32-bit address is in the X register. Pop the data stack. The address in the X register is not modified.

This store instruction is different from the “!” instruction in FORTH, which uses an address on top of the data stack.

Coding Example:

```
: !  ( n a -- ) tx stx ;
: 2! ( d a -- )  tx pushr stxp popr stx ;
```

STXP Store with X Register, Auto-Incrementing

Code:	13
Usage	Short Instruction
Stack Effects	(n -- ; X: a – a+1)
Carry	Restore from data stack

Function:

Store T into the memory location whose 32-bit address is in the X register. Pop the data stack. The address in the X register is then incremented by 1 to facilitate the next memory access. It is most useful in storing values to an array in memory.

Coding Example:

See the copying program shown in LDXP.

```
: 2! ( d a -- )  tx pushr stxp popr stx ;
```

TX Pop T to X Register

Code:	29
Usage	Short Instruction
Stack Effects	(a --)
Carry	Restore from data stack

Function:

Store T in the X register. Pop the data stack. The original contents in the T register are copied into the X register. This instruction initializes the X register so that it can be used to fetch data from memory or store data into memory.

Coding Example:

```
: +! ( n a -- )  tx ldx add stx ;
: 2! ( d a -- )  tx pushr stxp popr stx ;
: 2@ ( a -- d ) tx ldxp ldx ;
```

XOR Bitwise Exclusive OR

Code:	20
Usage	Short Instruction
Stack Effects	(n1 n2 -- n3)
Carry	Exclusive OR n1(32) and n2(32)

Function:

Pop S from the data stack and bitwise exclusive-OR it to the T register. All 33 bits in T are affected.

Coding Example:

To clear T to zero:

```
    DUP XOR cccccc cccccc
```

To generate a zero in T register:

```
    DUP DUP XOR cccccc cccccc
```

To generate -1 in T::

```
    DUP DUP XOR COM
```

```
::  < ( n n -- t )
   2DUP XOR 0<
   IF DROP 0< EXIT THEN
   - 0< ;;
```

XT Push X Register to T

Code:	25
Usage	Short Instruction
Stack Effects	(-- a)
Carry	Restore from X

Function:

Copy the contents of the X register to the T register. The original contents in the T register are pushed onto the data stack. With the XT and TX instructions, the X register can serve as a scratch pad to save and restore the contents of the T register.

Coding Example:

```
: SWAP ( n1 n2 - n2 n1 )
   pushr tx popr xt ;
: ROT ( w1 w2 w3 -- w2 w3 w1 )
   pushr pushr tx popr
   popr xt ;
```

Appendix B: eP32 eForth Commands

' <name>	-- xa	Find <name> and leave its execution address, xa.
-	w1 w2 -- w3	Subtract w2 from w1. w1-w2=w3.
!	w a --	Store w at a.
#	u1 – u2	Extract least significant digit from u1 and leave quotient, u2.
#>	w -- a u	Discard w, and leave address and length of number held in string buffer.
#S	u -- 0	Convert u to a number string below PAD buffer.
$" <string>”	-- a	Compile a string literal delimited by “. At run time, leave its address on stack.
$"|	-- a	Run time command of a string literal. Leave string address, a, on stack.
$," <char>	--	Compile a character literal.
$,n	a --	Compile a name field in header with string at a.
$COMPILE	a --	Compile a word whose name string is at a.
$INTERPRET	a --	Interpret a word whose name string is at a.
(<string>)	--	Ignore the comment string delimited by).
(CALL)	a --	Compile a subroutine call to address a.
(parse)	b u c -- b u delta	Parse next string delimited by c in buffer b, length u. Length of parsed string is delta.
*	n1 n2 -- n3	Multiply. n3=n1*n2.
*/	n1 n2 n3 -- nq	Leave quotient of (n1*n2)/n3.
*/MOD	n1 n2 n3 -- nr nq	Leave remainder, nr, and quotient, nq, of (n1*n2)/n3.
,	w --	Add w to parameter field of the most recently defined command.
.	n --	Display signed number with a trailing blank.
." <text>"	--	Compile a string literal <text>. At run-time display <text>.
."|	--	Run time command of . ".
.(<text>)	--	Display a string <text>.
.ID	xa --	Display name of a command at xa.
.OK	--	Display system OK message.
.R	n u --	Display number n right justified in a field of length u.
.S	--	Display the contents of data stack.
/	n1 n2 – nq	Division. Leave signed quotient of n1/n2.
/MOD	n1 n2 – nr nq	Division. Leave signed remainder, nr, and quotient, nq, of n1/n2.
: <name>	--	Begin a colon command of <name>.
;	--	Terminate a colon command.
?	a --	Display contents of memory at a.
?DUP	w -- w w | w	Duplicate w if it is not 0. Else no operation.
?KEY	-- c true | false	Return a false flag if no character is entered from keyboard. Else leave valid character and true.
?UNIQUE	a – a	If string at a is a valid command, display “redef” message.
@	a -- x	Replace address a by its contents.
@EXECUTE	a --	Execute word whose execution address is in address a.
[	--	Switch from compilation to interpretation.
[COMPILE] <name>	--	Compile command <name> in input stream. It compiles an immediate command.
\ <text>	--	Ignore <text> until end of line.
]	--	Switch from interpretation to compilation.
^H	a1 a2 a3 –	Process backspace. Decrement current character pointer, a3, if it

	a1 a2 a4	is greater than buffer address a1.
+	n1 n2 -- n3	Add n1 and n2.
+!	w a --	Add w to number at address a.
<	n1 n2 -- flag	True if n1 less than n2. Signed comparison.
<#	--	Start number conversion process.
=	n1 n2 -- flag	True if n1 equals n2.
>B	a b -- a+1 b+4 count	Unpack word string at a to byte string at b. Return a+1, b+4 and a count to unpack next word.
>CHAR	c – n	Convert character c to a valid character code.
>NAME	xa -- na | 0	Convert execution address, xa, of a command to its name field address. na. If failed, return 0.
>R	w --	Push top item to return stack for temporary storage.
0<	n -- flag	Return true if n is negative.
1-	n – n-1	Decrement.
1+	n – n+1	Increment.
2!	d a --	Store a double integer to address a.
2@	a – d	Fetch a double integer from address a.
2DROP	d --	Drop a double integer.
2DUP	d – d d	Duplicate a double integer.
4/	n – n/4	Divide by 4.
ABORT	--	Return to terminal interpreter, no error message.
ABORT"	--	Compile an error message. Execute abort" at run time.
abort" <string>“	flag --	If flag is true, abort and display an error message.
ABS	n -- u	Convert n to its absolute value, u.
accept	a u1 -- a u2	Accept text from keyboard into buffer at a, length u1. Return with a and actual length of text, u2.
AFT	a1 – a2	Start compiling an AFT-THEN structure in a FOR-NEXT loop.
AGAIN	a --	Terminate a BEGIN-AGAIN loop by compiling a branch to address a.
AHEAD	-- a	Compile a branch instruction. Leave its address on stack to be resolved later by THEN.
ALLOT	u --	Extend u bytes to parameter field of the most recent command.
AND	w1 w2 -- w3	Logical bit-wise AND.
B>	b a -- b+1 a	Pack a byte at b into least significant byte in a. Increment b.
BEGIN	--	Start an indefinite loop like BEGIN-AGAIN, BEGIN-UNTIL or BEGIN-WHILE-REPEAT.
BL	-- 32	Get ASCII code of a blank or space.
CHARS	c u --	Display character c u times on terminal.
CMOVE	a1 a2 u --	Move u bytes starting from address a1 to memory starting at a2.
CODE <name>	--	Define a new primitive comand.
COLD	--	First command executed after CPU powers up.
COM	--	Assemble a COM machine instruction.
COMPILE	--	Compile following command to parameter field of currently compiled word.
CONSTANT <name>	w --	Define a constant. At run-time, w is left on the stack.
COUNT	a -- a+1 c	Get one byte c from address a and increment a.
CR	--	Display a new line.
CREATE <name>	--	Create a new data array with <name>. No parameter field space is reserved.
DECIMAL	--	Set number base to decimal.

DIAGNOSE	-- 12 chars	Produce a string of “eForthMlSemi” to verify primitive commands.
DIGIT	u -- c	Convert number u to corresponding ASCII code.
DIGIT?	c base -- u flag	Convert ASCII code c to its corresponding number, u. If successful, return u and true. If unsuccessful, return c and false.
dm+	a u – a+u	Dump u bytes of memory starting at address a.
DNEGATE	d -- -d	Negate a double integer.
do$	-- a	Run time routine of $. Leave address of the following string literal.
DOES	--	Start compiling an interpreter for a new class of defining commands.
DOVAR	--	Run time routine for variables.
DROP	w --	Discard top of stack.
DUMP	a u --	Dump u bytes of memory starting at address a.
DUP	w – w w	Duplicate top of stack.
ELSE	--	Terminate a <true> clause, and start a <false> clause in IF-ELSE-THEN branch structure.
EMIT	c --	Display character c on terminal.
ERROR	a --	Display an error message at address a and abort.
EVAL	--	Evaluate (interpret or compile) input stream accepted into terminal input buffer.
EXECUTE	a --	Execute a command whose execution address is a.
EXIT	--	Terminate execution of a colon command.
EXPECT	a u --	Accept input stream into buffer at address a, length u.
EXTRACT	u1 base – u2 c	Extract least significant digit in u1, with radix base. Return quotient u2 and extracted character c.
FILL	a u c --	Fill an array at address a, length u, with byte c.
find	a va -- xa na | a 0	Search vocabulary beginning at va for a word whose name is at address a. If success, return execution address, xa, and name field address of command found. Else return a and false flag.
FOR	--	Start a FOR-NEXT loop.
FORGET <name>	--	Search dictionary for <name> and delete it and all subsequent commands from dictionary.
HERE	-- a	Get address of next available dictionary location.
HEX	--	Set number base to hexadecimal.
HOLD	c --	Add character c to number conversion buffer.
IF	--	Start an IF-ELSE-THEN branch structure. At run time, branch to ELSE or THEN if top of stack is 0.
IMMEDIATE	--	Add immediate bit to name of the command currently under compilation. An immediate command is executed by compiler.
KEY	-- c	Wait for an ASCII character c from the keyboard. KEY does not echo the character.
kTAP	bot eot cur c -- bot eot cur	Add a character, c, received from keyboard to string in terminal input buffer. bot is bottom of buffer, eot is end of buffer, and cur is pointer to current character in buffer. Process backspace.
LITERAL	w --	Compile number w as an in-line literal. At run-time, w is pushed onto stack.
M*	n1 n2 – d	Double precision multiply, d=n1*n2.
M/MOD	d n -- nr nq	Floored division. Return both remainder, nr, and quotient, nq.
MAX	n1 n2 -- n3	Return n3, the larger of n1 and n2.
MIN	n1 n2 -- n3	Return n3, the smaller of n1 and n2.
MOD	n1 n2 -- nr	Modulus, signed remainder of n1/n2.
NAME?	a -- xa na | a 0	Search dictionary for a command at address a. If successful, return its execution address, xa, and name field address, na. Else return a with a false flag.

NAME>	a – xa	Convert name field address, a, to execution address, xa.
NEGATE	--	Assembler machine instructions to negate top of stack.
NEXT	--	Terminate a FOR-NEXT loop. At run time, decrement index and repeat loop until index is 0.
NOT	w1 -- w2	Bit-wise one's complement.
NUMBER?	a -- n 1 | a 0	Convert a number string at address a to its value. If successful, return value n and true; else return a and false.
OK	--	Compile source text downloaded from terminal to file buffer, READBUF.
OR	--	Assembler OR machine instruction.
OVER	--	Assembler OVER machine instruction.
OVERT	--	Make the command last defined visible to interpreter and compiler.
PACK$	a1 u a2 – a2	Pack a counted string in address a1, length u to byte buffer a2.
PAD	-- a	Get address of a scratch pad area above dictionary of at least 84 bytes.
PARSE	c -- a u	Parse out the next string in terminal input buffer, delimited by character c. Return address a and length of parsed string u.
QUERY	--	Wait for a line of text from keyboard and place it in input terminal buffer. A line is terminated by carriage return or up to 80 characters.
QUIT	--	Return to terminal interpreter, no stack change, no message.
R@	-- n	Duplicate R register to top.
R>	n --	Pop return stack to top.
READ	--	Read text file from terminal into file buffer, READBUF.
REPEAT	--	Terminate a BEGIN-WHILE-REPEAT loop.
ROT	w1 w2 w3 -- w2 w3 w1	Rotate third item to top.
SAME?	a1 a2 u – a1 a2 f (-0+)	Compare two name strings at a1 and a2. Return 0 if identical. Return positive value if string1>string2. Return negative value if string1<string2.
SEE <name>	--	Decompile the command <name>.
SEND	a n --	Upload memory array at address a, length u, to host in Intel Hex format.
SIGN	n --	If n is negative, add minus sign to number conversion buffer.
SPACE	--	Display a space.
SPACES	u --	Display u spaces.
str	n – a u	Convert number n to a number string at address a, length u.
SWAP	--	Assembler machine instruction to swap top of stack.
TAP	bot eot cur c -- bot eot cur	Add a character c received from keyboard to string in terminal input buffer. botis bottom of buffer, eot is end of buffer, and cur is pointer to current character in buffer.
THEN	--	Terminate IF-ELSE-THEN branch structure.
TIB	-- a	Get address of terminal input buffer.
TOKEN	-- a	Get the address of next string parsed out of terminal input buffer.
TYPE	a u --	Display a string of u characters starting at address a.
U.	u --	Display unsigned number u with a trailing blank.
U.R	n1 u2 --	Display unsigned number u1 in a field of u2 characters.
U<	u1 u2 – f	Unsigned compare. Return true if u1<u2.
UM*	u1 u2 – ud	Unsigned double precision multiply. ud=u1*u2
UM/MOD	ud u -- ur uq	Unsigned double precision divide. Leave both remainder, ur, and quotient, uq.
UM+	u1 u2 – u3 carry	Double precision add. u3=u1+u2. Return carry also.

UNPACK	a b -- b	Unpack a packed string at a to b. String length is up to 255 characters.
UNPACK$	a b -- b	Unpack a packed string at a to b. String length is up to 31 characters.
UNTIL	--	Terminate a BEGIN-UNTIL loop structure.
VARIABLE <name>	--	Define a new variable. At run-time, variable <name> leaves its address on stack.
WHILE	--	Start a true clause in BEGIN-WHILE-REPEAT loop structure. At run time, repeat true clause while top of stack is non-zero.
WITHIN	u ul uh – flag	Leave true if ul <= u < uh. Else leave false.
WORD	c -- a	Get a string delimited by character c from the input stream and leave it as a counted string at address a.
WORDS	--	Display all words in dictionary.
XOR	--	Assembler XOR machine instruction.